Two-Dimensional Fourier Processing of Rasterised Audio

Chris Pike

cwp500@york.ac.uk

June 13, 2008

Abstract

A comprehensive software tool has been developed in Matlab to enable two-dimensional Fourier analysis and processing of audio, using the raster scanning process to obtain a two-dimensional signal. An understanding of the two-dimensional frequency domain representation has been acquired, and appropriate techniques for accurate analysis via rasterisation investigated. Audio transformations in the two-dimensional frequency domain were explored and some interesting and useful results obtained. The rasterisation method is however limiting when performing two-dimensional Fourier analysis and processing of audio.

ACKNOWLEDGMENTS

I would like to thank:

- Jez Wells for his constant support, understanding and enthusiasm.
- Yongbing Xu for his useful comments after the initial project report.
- Jason Gallicchio for his friendly and helpful advice on complex data to colour conversion.
- Christopher Penrose for his innovative perspective on audio signal processing.

Contents

Chapter 1

Introduction

Transform techniques provide a means of analysing and processing signals in a domain most appropriate and efficient for the required operation. There are many different transform techniques used for audio processing [27], however these techniques all have limitations and no method is suitable for all types of signals. There is continuous research effort into expanding, adapting and improving the available transform techniques for audio applications.

There has been recent work in the rasterisation of audio and sonification of images using raster scanning [39], which provides a simple mapping between one-dimensional (1D) and two-dimensional (2D) representations. A 2D audio representation allows the application of transform techniques in two dimensions, such as those used regularly in image processing. The use of 2D transform techniques for audio has seldom been explored [23] and may produce interesting and useful results.

1.1 Project Aims

This project is an investigation into the application of 2D transform techniques to audio signal processing, particularly focusing on the use of the 2D Fourier transform. It aims to discover what information can be gained about an audio signal by analysing it in the 2D Fourier transform domain and investigate what methods of transform data processing can be used to provide musically interesting sound modifications.

In the initial stages of the investigation, the following key aims can be identified:

i. Using raster scanning, obtain a 2D representation of an audio signal that can be visualised as a meaningful image.

ii. Apply 2D Fourier transform techniques to audio data.

iii. Develop clear understanding of the information given by a 2D Fourier transform of audio.

iv. Identify musically interesting transformations of audio using 2D Fourier transform data.

v. Make these processing techniques accessible to composers and sound designers.

In order to achieve these aims a large element of software development will be required during the investigation. The main objective of the project will be to produce a software tool that enables analysis and transformation of audio using data in the 2D Fourier transform domain. It should provide an insight into any signal features that can be characterised using the 2D Fourier transform and allow musically interesting manipulations of audio using the 2D Fourier domain data.

1.2 Report Structure

The report will start with the relevant background information required as a precursor to the project work in section 2. This includes a survey of existing research that covers many aspects of the project, and similar ideas that have served as inspiration for the work. Section 3 will introduce the software tool developed during the project. It will cover the overall requirements and broader software engineering aspects of the tool development. The software implementation of 2D Fourier analysis and processing is saved for subsequent sections 4 and 5. These sections will also include the theoretical aspects of the work carried out. After the software tool has been fully described, an evaluation of the results of the analysis and processing of audio using the 2D Fourier transform will be given in chapter 6. The testing process is described in section 7 and the project conclusions and recommendations for future work are given in sections 8 and 9 respectively.

Chapter 2

Background Information

This section details the background information required for this project including a survey of relevant research literature. This review has served several purposes:

- Understanding the context of the project by examining techniques currently used for audio and image processing purposes.
- Developing an understanding of the techniques that will be used in the project.
- Understanding the similarities and differences in Fourier processing of images and audio.

2.1 Fourier Techniques

In the late 19th century, Joseph Fourier proposed a theorem that any periodic signal can be decomposed into a series of harmonically related sinusoidal functions with specified amplitude, frequency and phase offset [28]; this is known as the Fourier series.

The Fourier transform uses this concept for decomposition of a continuous-time signal into sinusoidal components; any signal of infinite extent and bandwidth can be perfectly reconstructed using an infinite series of sinusoidal waves representing all frequencies from 0 Hz to ∞ Hz. These sine waves form the Fourier transform spectrum, which represents the frequency content of the signal and is symmetric about 0 Hz for a real input signal. Using the polar signal representation, the Fourier transform spectrum can be divided into

magnitude (amplitude) and phase components, describing their values for the continuous range of frequencies.

Fourier analysis is extremely important in many areas of signal processing. In audio processing it provides intuitive representation of the signal content, since stationary sinusoidal basis functions cause the minimum region of excitation on the basilar membrane in the ear [24].

2.1.1 One-dimensional Discrete Fourier Transform

The discrete Fourier transform (DFT) is a more relevant tool in digital signal processing since it allows analysis of frequency characteristics for a discrete-time signal, such as a sampled audio waveform.

The spectrum of a discrete-time signal is periodic about the sampling frequency [17] and a signal that is periodic in the time domain has a discrete spectrum, since it is composed of only harmonically related sinusoids, placed at integer multiples of the fundamental frequency. The discrete Fourier transform views the signal under analysis as a complete cycle of an infinitely periodic waveform, hence the signal is discrete in both time and frequency. The transform process is invertible using the inverse discrete Fourier transform (IDFT), which recreates the time-based signal from its frequency points. The formulae for the DFT and its inverse, of a signal of length N samples, are:

$$X[v] = \sum_{n=0}^{N-1} x[n]\, e^{-\frac{2\pi j}{N} vn} \qquad v = 0, 1, \ldots, N-1 \tag{2.1}$$

$$x[n] = \frac{1}{N} \sum_{v=0}^{N-1} X[v]\, e^{\frac{2\pi j}{N} vn} \qquad n = 0, 1, \ldots, N-1 \tag{2.2}$$

The DFT is a discrete-time representation of the exponential Fourier series. It produces the same number of frequency samples as there are time samples; this is N, the size of the DFT. The actual frequency that each point in $X[v]$ represents is given by the formula:

$$f_v = \left(\frac{v}{N}\right) f_s \tag{2.3}$$

Where f_s is the sampling frequency. The DFT only analyses the signal at these discrete frequencies and it is likely that the actual frequency components of the signal will fall

between these analysis bins. When a frequency component is not located directly in the centre of an analysis channel its energy will be spread across adjacent bins [27]. This is a major source of uncertainty in the DFT analysis, the frequency spectrum may indicate components that are not actually present in the signal, and particularly when more than one component is between analysis channels.

The range of frequencies covered is from 0 Hz to $\frac{N-1}{N} f_s$ Hz [27]. However the values in the frequency spectrum are reflected about $\frac{N}{2}+1$ due to the periodicity of the spectrum, so the DFT signal $X[v]$ contains redundant information for values of v above this point. Therefore the longer the duration of $x[n]$, the greater the resolution of the frequency analysis will be [36].

2.1.2 Fast Fourier Transform

The DFT was not efficient enough to be widely used in digital signal processing applications because it requires $\Theta(N^2)$ complex multiply & add operations. The fast Fourier transform (FFT), introduced in the 1960s [5], reduced the complexity of the operation to $\Theta(N\log_2 N)$ multiplications by exploiting the redundancy in the spectrum. This improved efficiency has allowed the use of the Fourier transform in many signal processing operations and real-time Fourier processing is now commonplace.

There are many different implementations of the FFT available, however MATLAB has built in functions, which employ the FFTW (the fastest Fourier transform in the West) algorithm [23, 24]. This is a very popular and efficient implementation of the DFT, which has the advantages of using portable C instructions that aren't hardware specific. It computes the DFT in operations for any input length N, unlike some implementations which are only efficient for a restricted set of sizes.

2.1.3 Two-dimensional Discrete Fourier Transform

The DFT of a two-dimensional array of $M \times N$ samples can be easily constructed by extending the one-dimensional DFT formulae [16].

$$X[u,v] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m,n]\, e^{-j2\pi\left(\frac{um}{M} - \frac{vn}{N}\right)} \qquad \begin{matrix} u = 0, 1, \ldots, M-1 \\ v = 0, 1, \ldots, N-1 \end{matrix} \tag{2.4}$$

$$x[m,n] = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} X[u,v]\, e^{j2\pi\left(\frac{um}{M} - \frac{vn}{N}\right)} \qquad \begin{matrix} m = 0, 1, \ldots, M-1 \\ n = 0, 1, \ldots, N-1 \end{matrix} \tag{2.5}$$

The equations for the DFT and its inverse have the same relationship in two dimensions as in one i.e. the inverse transform is the complex conjugate of the forward transform, divided by the number of points in the transform. The variables u and v are the frequency variables and the variables m and n are the time or spatial variables, depending on the type of signal represented.

The 2D DFT is obtained by performing the N-point DFT of each of the M rows of the array, so obtaining an intermediate $M \times N$ array of the results. The M-point DFT of each of the N columns of this array is then taken to give the final 2D DFT array. The process can also be done in the opposite order, columns then rows, and the same result will be obtained.

As with the 1D DFT, if the input $x[m,n]$ is real then the Fourier transform is conjugate symmetric and the magnitude spectrum is symmetric. This is shown by the equations:

$$F[u,v] = F^*[-u,-v] \tag{2.6}$$

$$|F[u,v]| = |F[-u,-v]| \tag{2.7}$$

Also the relationship of resolution in sampling intervals between the two representations is similar to the 1D DFT:

$$\Delta u = \frac{1}{M\Delta m} \tag{2.8}$$

$$\Delta v = \frac{1}{N\Delta n} \tag{2.9}$$

Where Δu and Δv are the two frequency domain sampling intervals, Δm and Δn are the spatial/time domain sampling intervals. Therefore the larger the 2D array is, the greater the resolution of the frequency points is or the smaller the frequency sample intervals

are. The precise meaning of the Fourier spectrum obtained from the 2D DFT is difficult to explain without the context of a signal. The understanding of the 2D spectrum is considered in different terms for images and audio, these are described in sections 2.2.7 and 2.3.1 respectively.

2.2 Frequency Domain Audio Processing

This section provides a review of current Fourier audio processing techniques, their applications and their drawbacks. The existing research into 2D Fourier processing is then reviewed.

2.2.1 Time-Frequency Representation of Audio

In 1946, Gabor [14] wrote a highly influential paper that founded the ideas of a time-frequency representation of audio and paved the way for many of the transform techniques now commonly applied to audio. Gabor proposed that time and frequency representations were just two extremes of a spectrum. Time based representations provide a detailed description of the temporal development of a signal whilst yielding no information about the frequency content. Fourier signal representations present the signal as a sum of infinite stationary sinusoids, which yield no information about time-varying events without converting back to the time domain. By using a time-frequency analysis framework, in which components are discrete in both time and frequency, a more intuitive analysis of the signal characteristics is possible. Non-stationary sinusoidal components are more familiar to the way we understand and listen to sounds.

2.2.2 Short-Time Fourier Transform

The short-time Fourier transform (STFT) is an adaptation of the Fourier transform that enables a time-frequency representation of audio. It operates by dividing the original long-time signal into many short frames, typically in the range of 1ms to 100ms in duration [27], which are shaped in amplitude using a chosen window function. The DFT can then be used to analyse the spectrum of each of these frames thereby producing a representation

of the variations in the spectrum of the signal over time.

The STFT equation [9] can be viewed as the DFT of an input signal $x[m]$ of arbitrary length M, multiplied by a time-shifted window function $h[n-m]$ of length N.

$$X[n,v] = \sum_{m=0}^{M-1} \{x[m]\, h[n-m]\}\, e^{-j(2\pi/N)vm} \qquad v = 0, 1, \ldots, N-1 \tag{2.10}$$

The output function $X[n,v]$ has N frequency points (indexed by v) for each of the time windows at a discrete-time index n.

It is common to overlap the time windows, since this enables a greater time-resolution. It also ensures equally weighted data when windows overlap at the half-power point of the window function. Serra [30] gave an alternative formulation of the STFT indicating the time advance of each window, commonly known as the hop size:

$$X[l,v] = \sum_{n=0}^{N-1} h[n]\, x[(lH)+n]\, e^{-j(2\pi/N)vn} \qquad \begin{matrix} l = 0, 1, \ldots, L-1 \\ v = 0, 1, \ldots, N-1 \end{matrix} \tag{2.11}$$

The number of spectral frames is given by:

$$L = \frac{M}{H} \tag{2.12}$$

Here M is the length of the input signal, $h[n]$ is the window function that selects a block of data from the input signal, l is the frame index and H is the hop size in samples. The frequency corresponding to each point v is given by equation (2.3), for each point n the corresponding time index is:

$$t_l = l \times (H/f_s) \tag{2.13}$$

Equation (2.11) describes the operation of the STFT, however it is quite often used for real-time signal processing, where the input signal length M is unbounded. In a real-time implementation the STFT would be performed on a frame-by-frame basis.

2.2.3 Window Functions

The window function is used to extract a small section from a long-time signal, so that a short FFT can be performed. The multiplication of the two functions (audio signal and window) in the time-domain corresponds to a convolution of the frequency spectra of these

functions, due to the theory of convolution. Therefore the window function will distort the analysed spectrum, introducing extraneous frequency components.

The most rudimentary window to use to extract a short section of a signal is a rectangular function, however this is likely to introduce discontinuities in the signal and will provide a distorted analysis, which is represented by high magnitude side lobes in the window's frequency response.

A smooth bell-shaped curve, such as a Gaussian function, will prevent spectral distortion due to discontinuities at the edges. It's frequency response has much lower side lobes, however a smoother curve also has a wider main frequency lobe. This means that the frequency resolution will be effectively reduced by the window's frequency response [26].

There are many different window functions commonly used, all with different spectral response characteristics. The MATLAB Signal Processing toolbox provides a window design and analysis tool that provides a detailed analysis of many common window types. The window function must be chosen according to the specific requirements of analysis or processing.

2.2.4 Resynthesis Methods

Resynthesis constructs a time-domain signal from the analysis data, which may or may not have been modified. There are a variety of different resynthesis techniques available [33]; a lot depend on a specific type of analysis but some simple examples are oscillator bank resynthesis and source-filter resynthesis.

The overlap-add resynthesis method is the inverse of STFT and therefore is commonly used to synthesise the STFT analysis data. The inverse DFT is applied to each frame of the analysis data to obtain a series of short-time signals. These waveforms can then be added together with the same overlap as in the analysis stage to reproduce the original time-domain signal. The inverse of the STFT equation (2.11) for overlap-add resynthesis is:

$$x\left[(lH)+n\right] = h\left[n\right]\frac{1}{N}\sum_{v=0}^{N-1} X\left[l,v\right] e^{j\frac{2\pi}{N}vn} \qquad \begin{matrix} l = 0,1,\ldots,L-1 \\ n = 0,1,\ldots,N-1 \end{matrix} \tag{2.14}$$

2.2.5 Phase Vocoder

The phase vocoder was invented in 1966 [13] and first applied to music applications in 1978 [22], it is now a popular tool in music signal processing because it allows frequency domain processing of audio. It is an analysis-resynthesis process that uses the STFT to obtain a time varying complex Fourier signal representation. However, since the STFTs analysis is on a fixed grid of frequency points, some calculation is required in order to find the precise frequencies contained within the signal. The phase vocoder calculates the actual frequencies of the analysis bins by converting the relative phase changes between two STFT outputs to actual frequency changes [9]. The phase values are accumulated over successive frames to maintain the correct frequency of each analysis bin.

The magnitude spectrum identifies that certain frequency components are present within a sound but phase information contributes to the structural form of a signal [2]. Phase information describes the temporal development of sound and enables the construction of time-limited events using stationary sinusoidal components.

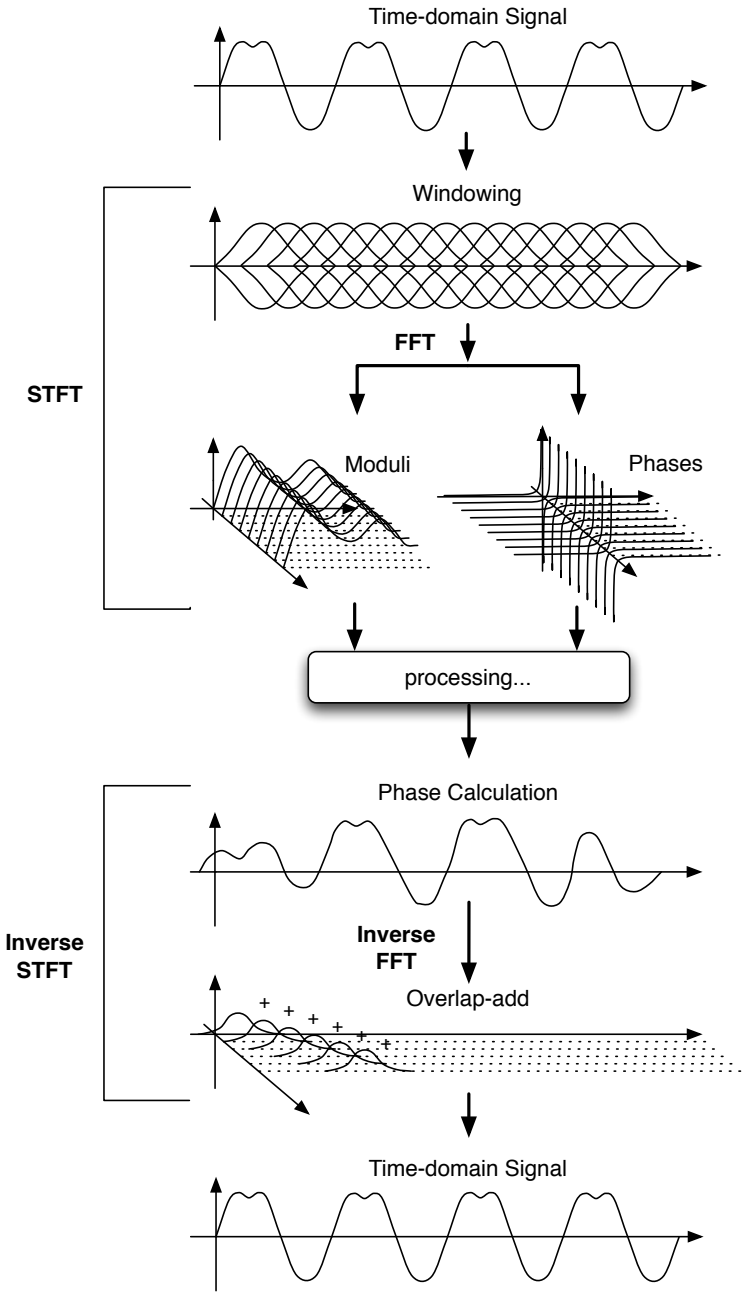


Figure 2.1: Phase Vocoder Functionality, after [8]

The ability of the phase vocoder to determine the deviation from centre frequency of each analysis bin allows effective stretching and compression of time without affecting pitch. The frequency changes are calculated on a different time basis and then the signal is

resynthesised using the inverse STFT. A diagram of the processes involved in a phase vocoder is shown in figure 2.1.

It is worth mentioning other sound transformations using the phase vocoder since these may serve as inspiration for two-dimensional transformations.

- Frequency-domain filtering - It is possible to alter the amplitude spectrum of a sound by multiplying it with the transform data of a filter function [29]. Time-variant filtering is possible and the spectral variations of one sound can be used to filter another, this is a form of cross-synthesis.
- Pitch shifting - It is commonly performed by time stretching then resampling the signal to get an output of the same length as the input. However Dolson suggested a method of formant preservation using the original magnitude spectrum to shape the shifted one [9]. There are more sophisticated methods available that better preserve the sound timbre [18].
- Timbre manipulation - The timbre of the sound can also be manipulated independently of pitch and time. The techniques in [18] enable effective shifting and copying of spectral peaks to create effects such as harmonising and chorusing.
- Stable and transient extraction - Using the frame-to-frame changes in frequency, stable and transient frequencies can be extracted from the audio signal [10].
- Dynamic range manipulation - Conventional dynamics processes such as compression, gating and expansion can be performed on each individual frequency band over time [10].
- Cross-synthesis - This involves extracting characteristics of one signal and applying them to another signal; it includes a variety of different processes [27].

Sound representation using the STFT and the phase vocoder has well known limitations. Its popularity is due to the fact that it enables musically interesting transformations of audio, but the time-frequency uncertainty principle is deeply embedded within its implementation. As a result, transforms using the phase vocoder often yield signals that are blurred/smeared in time and frequency [27]. Also due to the overlapping of frames, a single time-frequency element cannot be modified alone without causing signal discontinuities.

2.2.6 Other Analysis-Resynthesis Tools

Tracking phase vocoder (TPV) algorithms such as the McAulay-Quatieri algorithm [21] detect peaks in the STFT analysis spectrum and assign them to tracks that record amplitude and frequency variations. Each peak is assigned to its nearest track from the previous frame and the number of tracks can vary over time, thereby allowing the system to monitor short-time components of the signal. The tracks are used to drive a bank of many sinusoidal oscillators at the resynthesis stage.

This method of analysis-resynthesis can more readily handle input signals with time-varying characteristics and inharmonic frequency content than the phase vocoder. Also the TPV allows more robust transformations of the analysis data than the regular phase vocoder. A similar set of transformations can be performed with the TPV such as spectrum filtering, cross-synthesis and pitch-time transformation [27].

The Spectral Modelling Synthesis algorithm (SMS) [30] provides an extension to the TPV that analyses the signal in terms of deterministic and stochastic components. The peak tracking method is carried out and then the residual signal is analysed using the STFT. The SMS method is more effective at handling noisy signals.

2.2.7 Two-Dimensional Fourier Analysis of Audio

Two-dimensional Fourier analysis has very rarely been applied to audio signals but it could potentially lead to new signal representations and creative sound transformations. In the initial chapters of his thesis [23], Penrose outlined a two-dimensional audio analysis framework.

This framework uses a 2D Fourier transform to analyse the audio, however the samples must first be arranged into a 2D array. In order to do this Penrose uses the STFT, as in equation (2.11), giving an array of complex-valued time-frequency points $X[l, v]$. For a frequency v, the L components of $X[l, v]$ can be considered as a complex-valued time-domain signal. The signal $X[l, v]$ can be reinterpreted as N time-domain signals, each representing the changes in amplitude of a particular frequency v, over time. Taking the N-point DFT of these time-domain signals provides a pure frequency domain representation. The process

can be expressed by substituting equation (2.11) into the following equation:

$$Y[u,v] = \sum_{l=0}^{L-1} h[l]X[l,v]e^{-j(2\pi/L)ul} \qquad \begin{matrix} u = 0,1,\ldots,L-1 \\ v = 0,1,\ldots,N-1 \end{matrix} \tag{2.15}$$

This formulation of the 2D DFT differs from the one in equation (2.4) because the time frames of the STFT can overlap. Therefore the number of output points in $Y[u,v]$, which is $(L \times N)$, is not necessarily the same as the number of input points, M. The overlap of frames prevents the frequency resolution of both dimensions from being implicitly linked, giving more flexibility in the analysis. The inverse 2D Fourier transform process can be achieved by substituting equation (2.14) into the following equation:

$$X[l,v] = \frac{1}{N}h[l]\sum_{u=0}^{L-1} Y[u,v]e^{j(2\pi/L)ul} \qquad \begin{matrix} l = 0,1,\ldots,L-1 \\ v = 0,1,\ldots,N-1 \end{matrix} \tag{2.16}$$

The STFT framework presents a two-dimensional signal representation where one dimension denotes time at intervals of H/f_s seconds and the other dimension denotes frequency components at intervals of f_s/N Hz. The 2D Audio Analysis framework, on the other hand, presents two-dimensions of frequency information. One dimension is the same as in the STFT representation, frequency components at intervals of f_s/N Hz and the other gives a lower frequency analysis at intervals of f_s/LH Hz.

The intention behind this 2D DFT process in [23] was to achieve an analysis of the subsonic variations of audible frequencies in spectral form, hence Penrose refers to the two frequency dimensions as the audible and rhythmic dimensions. To clarify the concept, the value of Y at point (u,v) corresponds to the amplitude and phase of a sinusoidal signal at an audible frequency of $v \times f_s/N$, that varies in amplitude at the rhythmic frequency $u \times f_s/LH$.

2.2.8 An Application of Two-Dimensional Fourier Analysis of Audio

Speech enhancement using the 2D Fourier transform was implemented in [32]. Noise reduction filters common in image processing, such as the 2D Weiner filter were applied to noisy speech signals. However although this technique produced effective results for speech free segments of the audio, during speech the noise was still quite high and musical in timbre

since its energy was correlated to the speech spectrum. Although the author did not speculate, this may be to do with the different requirements of noise reduction in image and audio to produce an acceptable output signal. The author found that a hybrid technique of 1D & 2D filtering produced the most effective results, strongly suggesting that there are still advantages to be gained from 2D Fourier processing of audio.

2.3 Image Processing

A digital image is a two-dimensional representation of a sample array; often this is a spatially sampled version of a continuous physical image. Image processing is a well-established and highly developed research field [16, 25] in which two-dimensional transform techniques are applied in many areas. Therefore when exploring the use of two-dimensional transform techniques for audio, it is useful to develop an understanding of their use in image processing and how this relates to audio signals. However, whereas digital audio signals are sampled in time, images are sampled spatially with no timing information; we must bear this in mind when making associations between the two signal types.

2.3.1 Two-Dimensional Fourier Analysis of Images

To obtain the Fourier magnitude and phase information of an image, the equations given in section 2.1.3 should be used. When the spectrum of an image is to be displayed, it is common to multiply the image $x[m, n]$ by $(-1)^{m+n}$ which centres the spectrum making it easier to comprehend. In the frequency domain the spectrum is periodic so shifting by $M/2$ pixels up/down and $N/2$ pixels left/right will centre the point (0,0). During image resynthesis by the inverse Fourier transform, the same shift must be reapplied to the data to regain the original form. The range of values in a magnitude spectrum is often very large, so to achieve a more intelligible display a logarithmic scale of pixel values is used. Figure 2.2 shows a test image and its shifted, log-scaled Fourier magnitude spectrum, produced in MATLAB, to illustrate the display representations. The axes of the displays correspond to the discrete indices from equations (2.4) & (2.5).

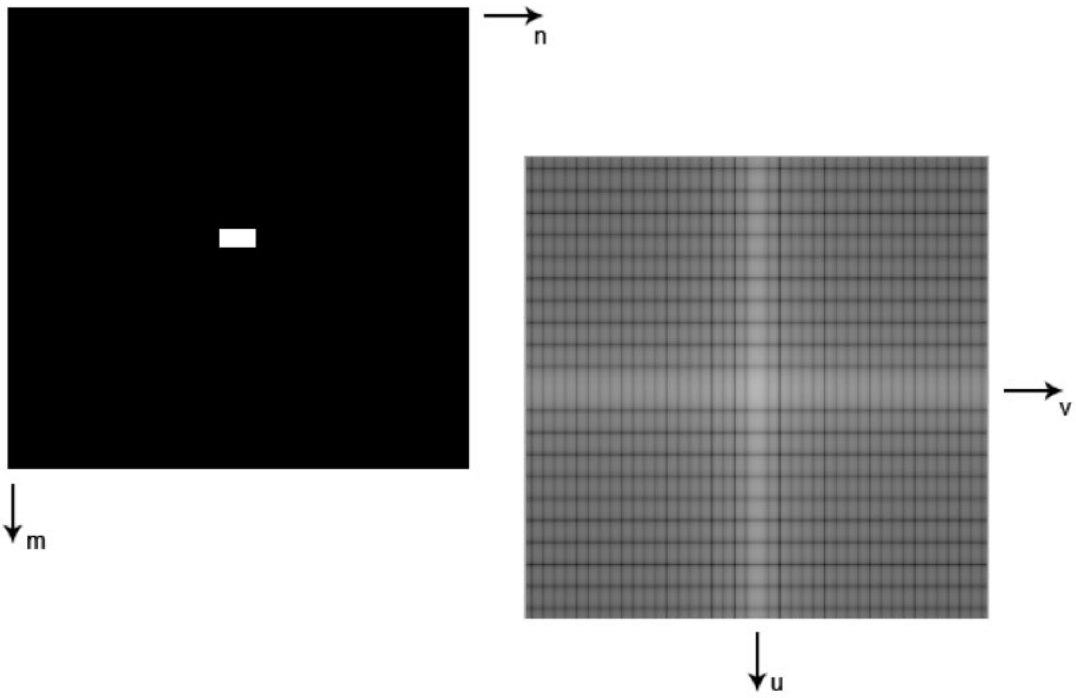
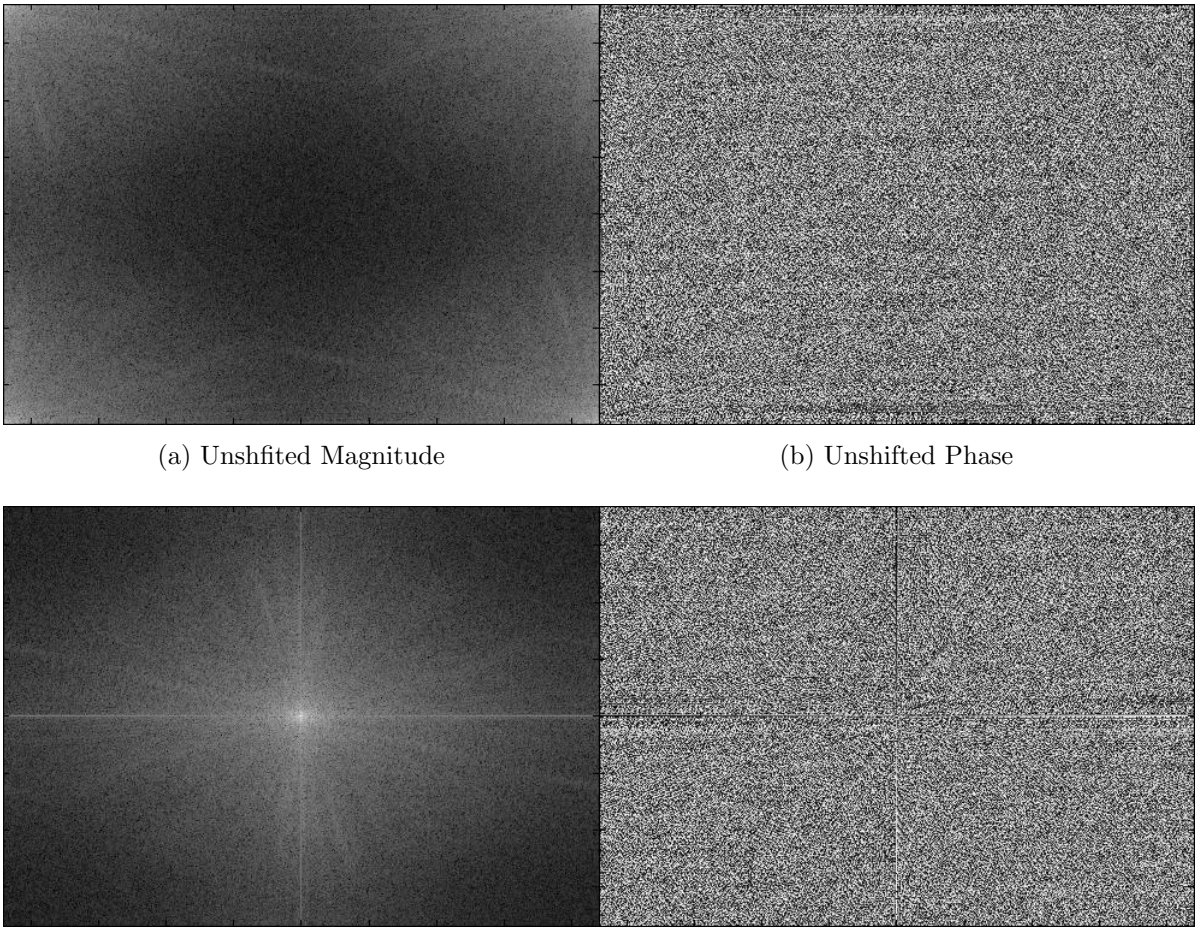


Figure 2.2: Image Display and 2D Fourier Magnitude Spectrum in Matlab, after [16]

Figure 2.3 shows the non-shifted and shifted displays of magnitude and phase information. Magnitude information becomes clearer when shifted but as you can see, the phase display is quite unintelligible and as with audio spectrum displays is often not shown. This does not however mean that phase information is not important in describing an image. As with audio (section 2.2.5), the phase information contains the structural information of the image. The importance of phase in the coherence of an image is demonstrated in [6], this website provides some simple images with their Fourier transform data to help develop an understanding of the relationship between an image and its Fourier data. The images in [6] demonstrate a method of displaying the magnitude and phase components together in a single display, where magnitude is the saturation and phase is the hue of the colour. Another web demonstration provides a Java applet to demonstrate the relationship between an image and its Fourier spectrum [15], allowing the user to modify the image or its spectrum and observe the effects on the other representation. This demonstration also shows magnitude and phase information in a signal display, this time using brightness to represent magnitude and chromacity for the phase.



(a) Unshfited Magnitude (b) Unshifted Phase

(c) Shifted Magnitude (d) Shifted Phase

Figure 2.3: Shifting of the 2D Fourier Spectrum, after [31]

Each point of the frequency domain representation $X[u, v]$ corresponds to a sinusoid, however since an image is based in the spatial domain rather than time, the understanding of this component is different. The point (0,0) corresponds to the DC value, which is average intensity value in the image. All other points refer to two-dimensional sinusoids, which have both frequency and direction [31]. This is demonstrated in figure 2.4, which shows an image with only a DC value and two images containing a sinusoidal waves of different frequencies and directions. Note that the frequency representation in this figure has not been shifted, so the point (0,0) is in the top-right corner and high frequencies are located near the centre of the spectrum whilst low frequencies are at the corners.

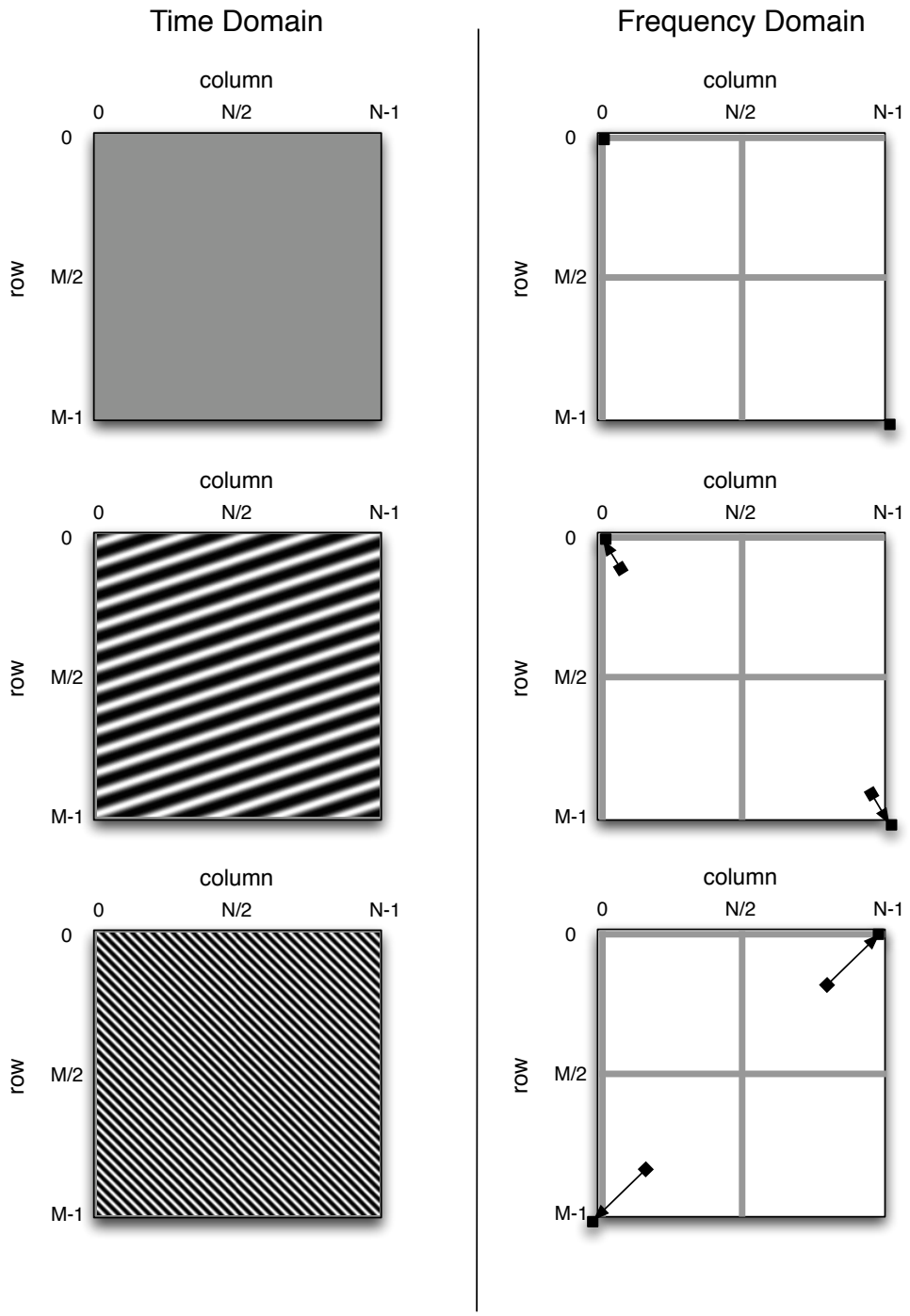


Figure 2.4: A Demonstration of Two-Dimensional Sinusoids in the Spatial and Frequency Domains [31]

Within more intricate real-life images, low frequency components correspond to the slowly changing intensity within an image such as the shadow of a wall and high frequency energy mostly corresponds to object edges, which cause rapid changes in intensity. As with transient components of audio signals, analysing an edge in terms of sinusoidal components requires many sinusoids of frequencies all along the spectrum. It therefore makes sense that an image with strong edges will produce prominent lines in the magnitude spectrum that run perpendicular to the edge angle. High frequency components are also present due to noise in the image.

2.3.2 Frequency Domain Image Processing

The 2D Fourier transform has various uses in image processing. Frequency-domain filtering is performed by multiplying a filter magnitude function with the magnitude spectrum of the image [16]; image filters have zero-phase. Any filter response can be used but commonly a low-frequency reduction results in image sharpening, a high-frequency reduction results in image smoothing and band-pass filtering allows image feature extraction. Ideal brick-wall filters cause undesirable ringing effects, which are more easily understood by thinking in terms of spatial convolution of the functions. Better results can be obtained using smooth filter functions such as Gaussian curves.

The Fourier transform of audio data converts from a confusing time-domain waveform into a more intuitive frequency spectrum representation, however in images the information is much more straightforward in the spatial domain and the Fourier representation is not very descriptive of the image. As a result, frequency domain filter design is not very useful [31]. The basic component of an image is the edge and edges are made of a wide range of frequency components. Filter design is more effective in the spatial domain where it is thought of as smoothing and edge enhancement rather than high & low pass filtering.

The most common use of the 2D Fourier transform for images is FFT convolution. A spatial image filter operates by convolving a neighbourhood kernel with the image, operating on each pixel one-by-one. For a large kernel this convolution operation becomes quite complex and it is more efficient to take the FFT of the image and the kernel, then multiply their spectra in the frequency domain. As previously stated, the DFT views the input function a periodic signal. In order to prevent aliasing distortion from corrupting the convolution,

both the image and the kernel must first be zero-padded [16]. The 2D DFT is also used for image restoration/noise reduction, data compression, motion estimation, boundary description and texture analysis purposes among others [16, 25].

2.4 Audio Feature Extraction

In order to obtain a useful image of an audio signal using raster scanning (section 2.5.1), the width should correspond to the period of the signal [39]. This identifies a requirement for pitch-detection techniques in this project; also for a longer rhythmic signal it may be beneficial to calculate the tempo to achieve a useful image display. A MATLAB toolbox for musical feature extraction from audio is described in [18], these tools are available for free use in the research community [20]. Pitch and rhythm recognition techniques are described in [26].

2.5 Graphical Representations

Image sonification and sound visualisation.

Another large research area relevant to this project is sound visualisation and image sonification methods. These techniques perform conversions between one-dimensional audio data and two-dimensional images, addressing the fundamental differences between the representations i.e. temporal and spatial. There are a huge variety of ways to map audio features to image features, the choice of method depends entirely upon what information needs to be portrayed. These techniques are applied to tools such as basic analysis displays, audio visualisation plug-ins and graphical composition/image sonification software.

A framework for the design of image sonification methods is defined in [38], which defines mapping geometries in terms of a pointer-path pair. It presents two opposing strategies, scanning and probing. Scanning methods have a predefined path for data sonification often with a fixed scanning rate. Probing methods allow variation of the speed and location of data pointer during sonification, which gives a much more flexible sonification. Scanning methods can easily be inverted in order to visualise a sound and so are the desired technique for this project. At any point in the image representation there could be multiple values

i.e. for a colour image, a sonification method can handle these values in any number of ways.

2.5.1 Raster Scanning

Raster scanning is a very common method of producing or recording a display image in a line-by-line manner, used in display monitors and also communication and storage of two-dimensional datasets. The scanning path covers the whole image area reading from left to right and progressing downwards, as shown in figure 2.5.

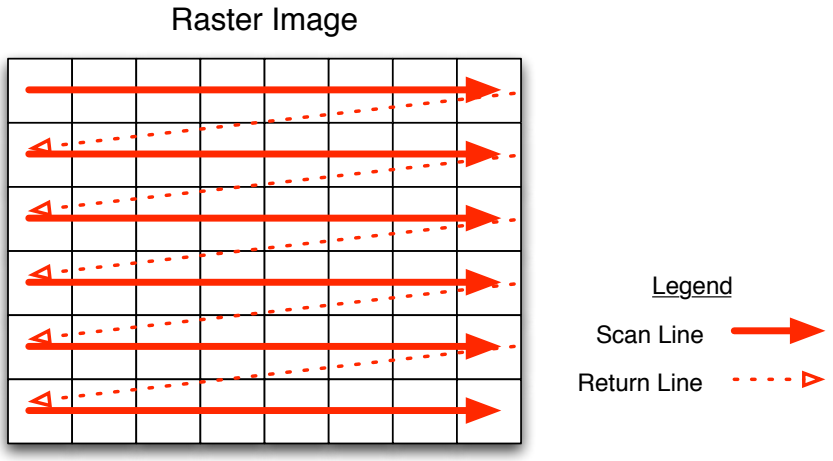


Figure 2.5: Raster Scanning Path

Raster scanning has recently been applied to audio visualisation and image sonification techniques [39]; it allows a simple one-to-one mapping between an audio sample and an image pixel. The process is perfectly reversible allowing it to be used as part of an analysis-synthesis system. It provides two dimensions of time as opposed to the STFT (time-frequency) and the 2D DFT (frequency-frequency).

Image sonification is performed using the raster scanning path, this process will be referred to as derasterisation. Sonified images produce a pitched time-developing sound as a result of the similarity between adjacent rows of pixels. Derasterised image textures produce similar audio textures and various visual filters were identifiable in listening tests [39].

Sound visualisation uses raster scanning to convert from a sound into an image, a process

which will be referred to as rasterisation. When the width of this raster image is correctly set to correspond to the sound's fundamental pitch period, the result shows temporal variations in pitch and timbre.

2.5.2 Spectrograms

The spectrogram is arguably the most popular method of sound visualisation. It produces a display from STFT analysis data, with time on one axis and frequency on the other. The intensity of the spectrogram at a certain point corresponds to the amplitude of a particular frequency at a certain time. The inverse spectrogram method is a popular scanning technique for image sonification, and if the parameters of the inverse STFT are the same as the analysis parameters, a sound can be recreated from its spectrogram.

2.5.3 Synthesisers & Sound Editors with Graphical Input

It is possibly to perform sound transformations by directly modifying the spectrogram. There are two notable software applications, Metasynth and Audiosculpt that allow sound design and modification using spectrogram editing and inverse spectrogram sonification. There is a large range of sound transformations possible with these techniques, such as directly adding and removing components to change the sound timbre and time-pitch scaling of data. The application of image processing effects leads to some interesting changes in the sound also. A list of spectrogram transformations available in Metasynth is given in [27].

2.6 Matlab Programming Environment

All investigation and software development has been carried out using Matlab. It integrates computing, visualisation and programming facilities into a single environment and therefore it is very well suited to an investigative programming project such as this. Software development in Matlab is often much quicker than a lot of other languages, and its use allowed this project to progress much further than would otherwise have been possible. Its basic data element is an array that does not require dimension specification unlike scalar

languages such as C. This allows much more flexibility. It's functionality focuses on matrix computation and so is well suited to signal processing applications. The software package provides a comprehensive set of pre-programmed Matlab functions (M-files), all of which are well documented. These functions span many categories including mathematics, data analysis, graphics, signal processing, file I/O, and GUI development.

Matlab is an interpreted programming language [37], the program is compiled to machine code as it is run and each line of code is reinterpreted each time it is read. This makes computation a lot slower than compiling languages such as C and Java, however the software tool for this project does not need to operate in real-time at audio rate. The run-time compiler allows quick code development and prototyping, since there are no compilation issues. Matlab also provides debugging tools and a command line which is useful for testing new functions and algorithms.

Chapter 3

Introduction to the Matlab Tool

Most of the project work was performed within the Matlab programming environment, investigating 2D Fourier techniques and incorporating them into a GUI based software tool. The experimentation and software implementation were carried out simultaneously throughout the project so that the GUI tool could provide a means of easy interaction with and observation of signals as new techniques were investigated. By the end of the project, the aim was to sufficiently develop the software tool to make 2D Fourier audio processing techniques accessible to both engineers and composers.

This chapter introduces the software tool which forms the basis of the project. It will describe the overall design and functionality, as well as the development process used.

3.1 Requirements of the Software Tool

The software tool needed to encompass all of the functionality required to meet the aims of the project, given in section 1.1. These aims yield the software requirements that ensure that a useful and useable tool is produced.

3.1.1 Signal Representations

One of the main aspects of this project was to investigate different representations of audio data and find out what each offers. There were three signal representations required

explicitly in the project aims; the audio signal and the raster image in the time domain, and the 2D Fourier spectrum in the frequency domain. It was decided that all signal representations should be displayed in the software tool, to help gain an understanding of the 2D Fourier domain properties by comparison.

The time domain audio signal is the basis from which all other representations are obtained. The signal can be displayed graphically to allow waveform inspection but it was most important to have an audio player in the software tool that would allow playback over a loudspeaker. Although the software tool displays signal representations visually, the input and output of this signal processing is sound.

The raster image is the 2D time domain representation of the audio signal obtained by rasterisation. This image is an intermediate form between the audio waveform and the 2D Fourier spectrum, but it also provides insight into signal variations over time that are hard to visualise in a 1D waveform display.

The 2D Fourier spectrum is the key signal representation in the software tool since it is the focus of the analysis and processing in this project. There is a large amount of data in the 2D Fourier representation, since each point in the 2D matrix has both magnitude and phase values, the display needed to clearly portray this data.

The 1D Fourier spectrum has also been included in the software tool as it is the more familiar representation of the frequency content of an audio signal and it would help users to grasp the concept of the 2D Fourier spectrum by comparison. It also aids general audio analysis and the understanding of the relationship between 1D & 2D Fourier representations. The 1D Fourier representation is obtained from the audio data using the FFT and is then decomposed into magnitude and phase components. Both of these needed to be obtained and represented within the software tool.

3.1.2 Analysis Tools

In order to allow proper investigation the software needed to provide several tools that would allow accurate analysis and aid inspection of the various representations of the signal. All plots needed proper headings, axis labels and scales to describe the data on display. Provision of plot tools such as zooming and panning, as well as a data pointer that displays the precise value at a specific point in the plot, was also important for thorough

analysis. Another required feature was a display of time-domain signal statistics such as range, mean and length, which would help characterise the signal.

Section 2.5.1 indicates that initial signal analysis is required to obtain a valid raster image by setting the appropriate image width. It was assumed that this would affect the 2D Fourier transform data and so the software tool also needed to allow definition of the initial settings for analysis. The user cannot be expected to know the signal characteristics such as pitch and tempo so feature extraction algorithms were also required within the software as part of this initial analysis.

3.1.3 Signal Transformations

The project includes an investigation of a variety of audio transformations obtained by manipulating the 2D Fourier domain data. The software had to incorporate these processes and allow the user to apply them to the audio signal.

Initially some key requirements of the transformation capabilities were decided. It had to be possible to adjust parameters of the processes once applied, so the processing could not destructive. Chaining of multiple signal transformations was required to provide a useful and flexible creative tool. Finally, the ability to switch between the unprocessed and processed signals was an important requirement both for the theoretical analysis of 2D Fourier transformations and as a reference during creative or experimental transformation of audio.

3.1.4 Additional Tools

It was important to consider any further tools that may be required by users, either composers or audio analysts. Data input and output is an important feature that needed consideration. It was vital that audio files could be imported and exported, for the purposes of the project investigation and also for creative users. It was also worth considering that a composer might want to store a signal along with its analysis settings and transformation processes, so that work could be continued over multiple sessions or shared with other users. One further requirement was the ability to save the software tool's display of the different data representations as an image file, which would be beneficial to the creation

of this report.

3.2 Software Development Process

Due to the investigative nature of this project, the software could not be produced using formal engineering methods. It was only possible to produce a specification of the general requirements of the software, since little was known initially about the properties of the 2D Fourier transform or the possibilities of using it creatively. It was decided that the most efficient process would be to develop the software gradually, treating it as a prototype application and constantly extending its capabilities. Within the time limits of the project it would not be possible to do the investigation and software engineering sequentially, and still produce a useful product.

The development process can however be divided into two distinct phases, both of which had a loosely predefined structure. The first phase was to develop a complete analysis tool, without which the properties of the 2D Fourier transform could not be properly investigated or understood. This then allowed a more informed investigation into 2D Fourier audio processing in the second phase; developing algorithms based on an understanding of the underlying theory and with some preconception of the results.

The details of implementation are omitted here, instead focusing on the path followed to reach the final software tool. The later parts of this section and the following sections 4 and 5 provide a more in depth guide to software implementation.

3.2.1 Analysis Phase

The analysis phase of software development was the longest, since the software tool had to be designed and restructured as the understanding of the 2D Fourier audio representation improved. Initially it was important to become fully aquainted with the Matlab environment and its capabilities; and then to produce the core functions required to load and display the data, so that from the earliest possible stage a useable tool was in operation.

The first step was to convert between the four required signal representations (section 3.1.1). The initial GUI and data structure designs were then produced, and a Matlab

figure was created to display the data plots. A simple audio player and import/export operations for the audio and images were also added at this point.

Once this initial version of the analysis tool was running, it was possible to load data and attempt to analyse it. This highlighted more features of the application that needed development, and the usability of the tool was constantly being improved. The data plots were enhanced, adding more flexibility and information, especially the 2D Fourier spectrum. The data structure underlying the software tool had to be restructured occasionally and the GUI was developed and adjusted as new functionality was introduced.

The focus of development then shifted to the initial analysis of the data to define the settings for audio rasterisation. Prior to this it was necessary to be able to import raster images and 2D spectra as image files as part of the investigation, however this functionality was not required in the later stages. This pre-processing stage was only designed to analyse an audio input. Feature extraction methods were incorporated into the software to automatically determine the raster image width, dependent upon the chosen analysis method.

Throughout this development, a variety of different audio signals were imported into the software to test the analysis process and find areas for improvement. More complex functions were tested formally before being incorporated into the tool, the details of testing are given in section 7. This experimentation led to further improvements to the initial signal analysis and tools. At this point, loading and saving of application data was added, which allowed the user to retain analysis settings with a signal.

The last part of this phase was to redesign the initial analysis capabilities of the software tool, adding more flexibility and presenting the user with a pop-up window of options, as described in section 4.5.

3.2.2 Processing Phase

This development phase was less incremental than the analysis phase. Once the initial structure for applying processes was in place, it was a simple process to make a new transformation available. The data structure had to be extended to store the processing information and retain the original signal data. The GUI and functionality for adding and removing processes had to be designed, and a generic process structure planned. Then it

was simply a case of creating processes one by one, testing them as they were built and occasionally going back to improve or correct the implementation. The loading and saving of application data also had to be extended to include processing parameters.

3.3 Graphical User Interface Design

Matlab offers two different methods for designing GUIs; the GUIDE (Graphical User Interface Development Environment) tool or programmatic development. GUIDE allows interactive design of a GUI and automatically generates an M-file containing initialisation code and the framework for the GUI callback functions. The other option is to specify the layout and behaviours entirely programmatically; each component is defined by a single function call and the callback functions have to be manually created.

3.3.1 Main Window

The main window for the 2D Fourier tool was designed using the GUIDE tool, because it allowed easy and immediate GUI development. It also provides a visual design interface so that design and development can be combined. The layout of the main window focuses around the graphical displays of the four data representations (section 3.1.1). They have been positioned to separate both the time and frequency domains, and the one and two dimensional representations as shown in figure 3.1. GUIDE saves the GUI design as a FIG-file and the M-file of automatically generated code, the main window is defined by `app.fig` and `app.m`.

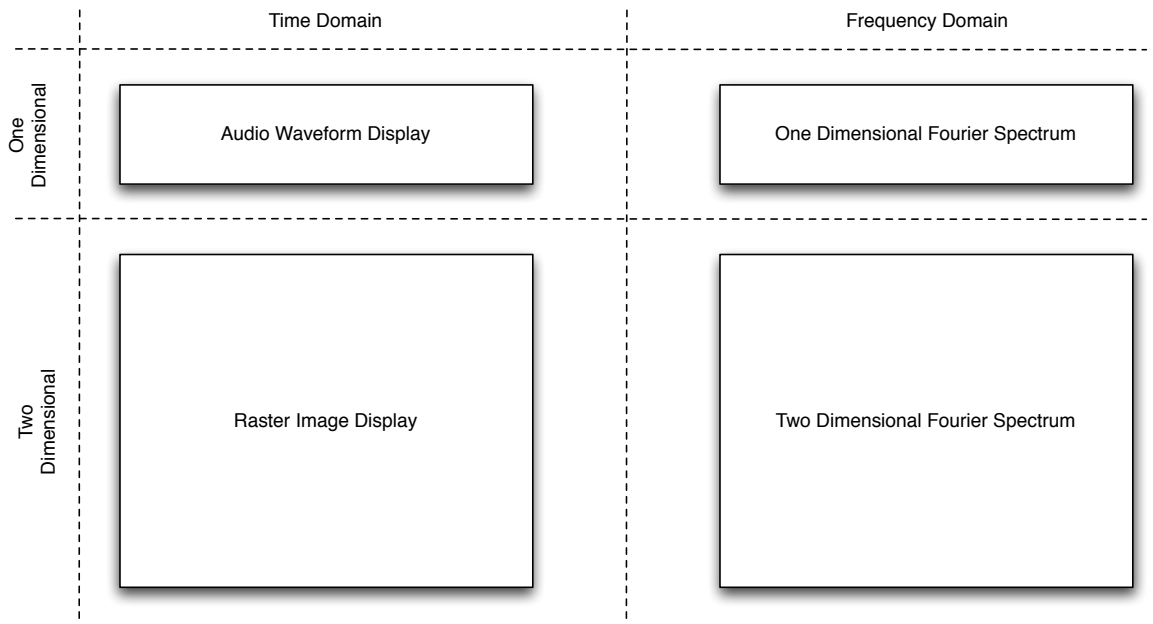


Figure 3.1: Plot Layout for User Interface

The GUI for the main window is displayed in figure 3.2. Below the four data plots there are several controls. On the right-hand side there are two panels containing the audio player (section 3.7) and the 2D spectrum display controls (section 4.4). On the left of the audio player are the frame selection buttons that allow the user to select which frame of data is displayed in the plots. The 'Processing' button opens the processing pop-up window that provides access to 2D Fourier signal processing; this is described in chapter 5. The 'Info' button opens a pop-up window that displays several signal properties and the 'Reanalyse' button opens a pop-up window that allows the user to adjust the initial analysis settings and redisplay the signal, these features are described in chapter 4. The main window also has a custom menu and toolbar, which provides access to data I/O, settings and analysis tools; the implementation is described in section 3.5.

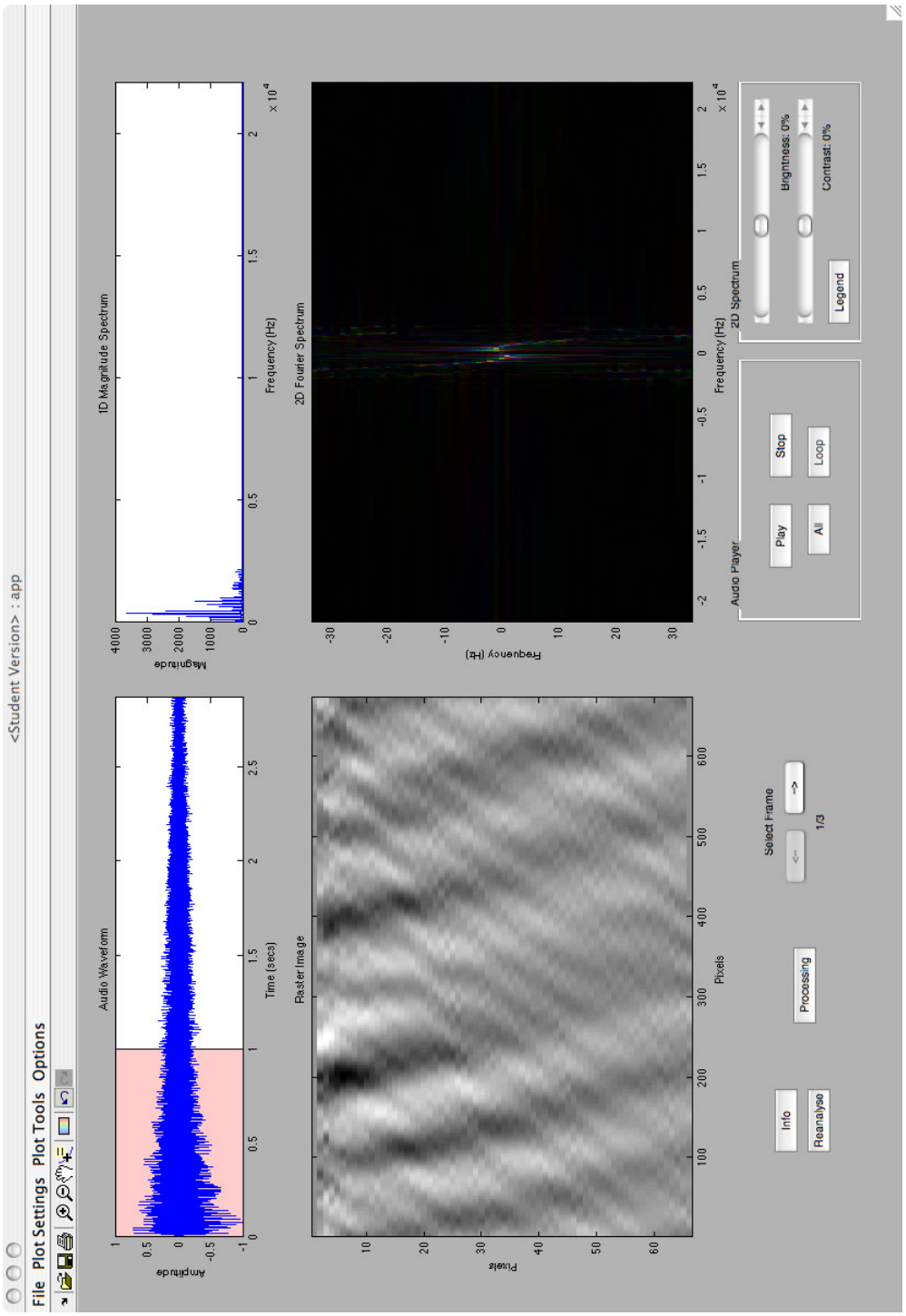


Figure 3.2: Main Software Tool GUI

3.3.2 Additional Windows

Additional windows within the software were designed programmatically, since precise control over the functionality and layout made it easier to interact with the main GUI. Most of these windows only required a collection of simple GUI components such as static text, text edit, push buttons and drop-down menus. The main aim with the layout of these GUIs was to have well spaced and aligned components that were logically grouped according to the information they portrayed.

3.4 Data Structures and Data Handling

The software tool handles a large amount of data, the data structures and the manner in which the software handled this data were important aspects of the software design. Matlab provides a mechanism for associating data with a GUI figure, which can be written to or read from at any time using the `guidata` function. Each graphics object in Matlab has a unique identifier called a *handle*, which can be used to adjust/obtain the object's properties and is used to associate data with a GUI figure, whether using the figure's handle or that of a child object. Only one variable can stored in GUI data at a time, so to store more information a structure is used.

Matlab's GUIDE tool (section 3.3) uses the GUI data to create and maintain the `handles` structure, which contains the handles of all components in the GUI. This structure has been extended in my software to contain all program data organised into relevant substructures, as shown in figure 3.3. The `handles` structure also contains the audioplayer and various plot tool objects, a matrix representing a unit circle (see section 4.8.3) and some additional Boolean variables used to define program behaviour. The M-file `app`, which was automatically generated by GUIDE, contains the function `app_OpeningFcn` which allows this program data to be initialised with the default settings just before the figure is loaded.

A description of each of the sub-level structures is given in table 3.1, however the details of their contents and functional role will be covered in later sections. The top level structure `handles` also contains a cell array named `proc_names`, which holds strings of the names of all processes described in chapter 5.

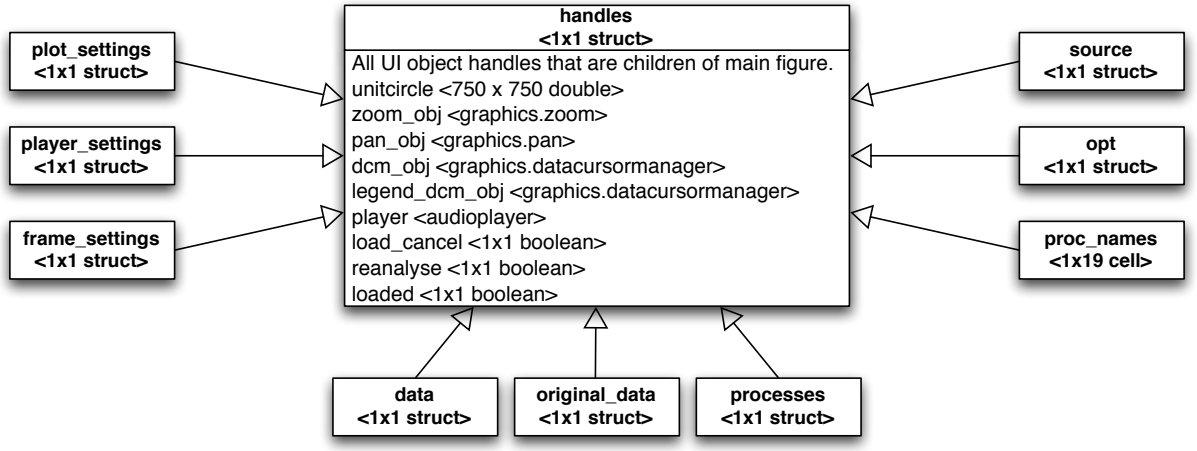


Figure 3.3: Top Level Data Structure

Structure Name	Description
`data`	Contains the four representations of the signal data along with any parameters that define the data and its representations. If any 2D Fourier processing has been applied then the signal data represents the output of the processing chain.
`original_data`	A copy of `data` before any 2D Fourier processing has been applied to the signal.
`processes`	The chain of 2D Fourier processing operations with all parameters that define them.
`plot_settings`	All variables that define the display options for the data plots in the main GUI.
`player_settings`	Variables used to control the operation of the audio player.
`frame_settings`	Data that allows individual frames of the signal to be displayed separately.
`source`	File path, name and extension of the data source and file paths for data I/O.
`opt`	Any other program options. Only the 'auto-normalise' option is currently in this structure.

Table 3.1: Description of Sub-level Data Structures

When a temporary GUI window is created, such as the analysis settings window (section 4.5), the object handles for the figure and all of its components are stored within a sub-structure of handles which is deleted when the figure is closed.

3.4.1 The `data` Structure

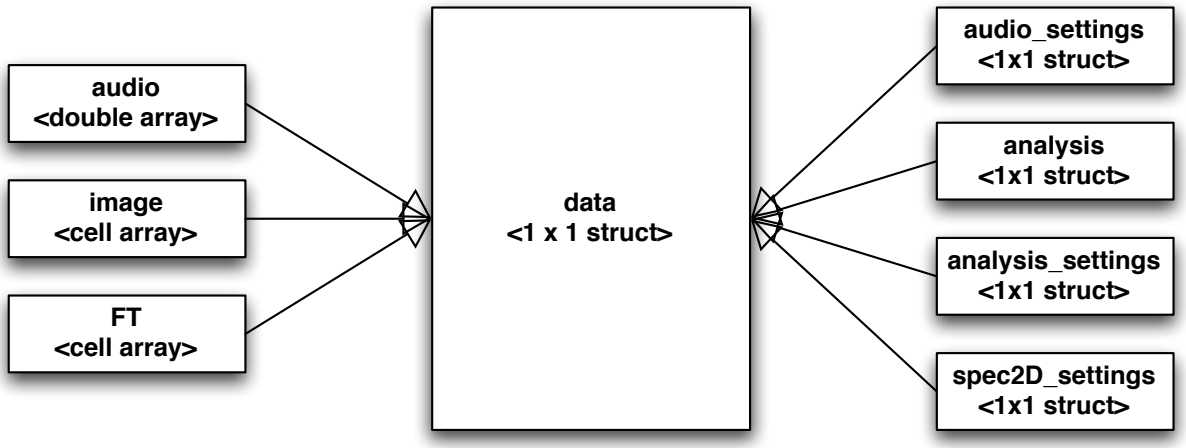


Figure 3.4: The `data` Structure

This structure contains the signal data and all related parameters. On the left hand side of figure 3.4 are the various representations of signal data that need to be stored. The audio data is contained within a one-dimensional array. The other representations, the raster images and the Fourier data, are stored within cell arrays since there can be more than one matrix of data for each. It is worth noting that the 2D frequency domain data is stored in the complex representation returned by `fft2` not the conventional polar representation, this is to allow flexibility in the display of this data. Each cell will contain a 2D array of data and this can be of any size, which is the main benefit of using cell arrays in this way.

On the right hand side of figure 3.4 are four structures containing parameters concerning the signal and its representations; they are described in table 3.2.

Structure Name	Description
`audio_settings`	Contains the sample rate and bit depth parameters, obtained from the source file with `wavread`, and also the duration in seconds of the audio data.
`analysis_settings`	This structure contains the direct parameters of the initial analysis options menu (section 4.5).
`analysis`	The subsequent parameters from the initial analysis that define the 2D representations, including frame size and number of frames, pitch related information and image width.
`spec2D_settings`	This structure contains the parameters concerning the conversion between raster image and 2D spectrum. The amount of padding applied and the maximum magnitude of the Fourier data.

Table 3.2: Description of Sub-level Data Structures

3.4.2 Callback Functions

The data flow within the software is event-driven, based on the interaction between the user and the GUI components. Matlab GUI components allow the use of callback functions to define their interactive behaviours. Listing 3.1 shows the definition of a generic object callback and the corresponding function. It also demonstrates the use of GUI data to pass variables into the callback function without explicitly naming them as arguments. When GUIDE is used the generated callback functions receive the `handles` structure as an argument automatically, however all GUI windows but the main one were developed programmatically and so their callbacks use the `guidata` function to obtain the `handles` structure.

```
guidata(component_handle, arg3);
set(component_handle, 'Callback', {@function_name,arg1, arg2, ...});

function function_name(source_handle, eventdata, arg1, arg2, ...)
  arg3 = guidata(source_handle);
  ...
end
```

Listing 3.1: Defining a GUI Object Callback and Using GUI Data

3.4.3 GUI Interaction

Most of the peripheral GUI windows were concerned with input/adjustment of parameters for analysis and processing algorithms. Often it was not just a case of obtaining a few numbers and returning them, many program variables were used in calculations to update several display objects or set several variables upon data entry. Each of these GUIs was a Matlab figure object and so could have its own GUI data, which enabled a local copy of the `handles` structure to be stored. In this way the program data could be edited without losing the original values and then the main figure's GUI data could be overwritten at a point decided by the program or the user. This method of data handling made the software tool much more flexible.

3.5 Menu and Toolbar Design

Matlab GUI figures allow the use of menus and a toolbar, which is useful when designing a software application. There are default menus and a toolbar that provide a lot of useful functionality such as access to plot tools but they also allow unnecessary and often undesirable operations such as insertion of GUI objects and saving of the figure file, which would overwrite the GUIDE figure layout. It was therefore necessary to develop custom menus and a custom toolbar that incorporated the required functionality from the defaults plus any additional functionality specific to the software tool.

The menus and toolbar can be seen in the main GUI display in figure 3.2. There are four menus:

- The 'File' menu, which contains the application specific data I/O operations load, save, import and export (section 3.6).
- The 'Plot Settings' menu, which allows the user to adjust the display settings for both the Fourier spectrum plots.

- The 'Plot Tools' menu, which provides the use of the zoom, pan and data cursor tools. The functionality of this menu was taken from the useful aspects of the default menus.
- The 'Options' menu, which allows the user to adjust the options stored in the `opt` data structure.

The toolbar allows quick access to some of the most commonly required menu options as well as some additional functions. Table 3.3 describes the function of each push button on the toolbar, dividers are used to group related buttons.

Name/Tooltip	Description
Load Data	Allows a signal to be loaded with its analysis settings, as in the 'File' menu.
Save Data	Allows a signal to be saved with its analysis settings, as in the 'File' menu.
Export Display	This button lets the user save the main GUI display to an image file.
Zoom In	Toggles the Zoom tool on and off with the direction set to 'in', as in the 'Plot Settings' menu.
Zoom Out	Toggles the Zoom tool on and off with the direction set to 'out', as in the 'Plot Settings' menu.
Pan	Toggles the Pan tool on and off, as in the 'Plot Settings' menu.
Data Cursor	Toggles the Data Cursor tool on and off, as in the 'Plot Settings' menu.
2D Spectrum Legend	Displays the 2D Fourier spectrum legend, as the 'Legend' button in the main GUI figure does.
View Original Signal	These buttons are grouped to toggle between displaying the signal data as loaded and analysed from the source file and the signal data after the 2D spectral processing chain has been applied. This is discussed in more detail in section 5.1.6.
View Processed Signal	

Table 3.3: Description of Toolbar Buttons

3.5.1 Implementation of Custom Layout

The custom menu was designed using GUIDE's graphical menu layout tool. The callback functions for each menu item were automatically generated as child functions within the `app` M-file and their functionality was inserted. The toolbar, on the other hand, had to be designed programmatically. The function `create_toolbar` initialises the toolbar object and all of the components on it. It is called at the start of `app_OpeningFcn`, which also calls `toolbar_callbacks` to define the callback functions for each component on the toolbar.

3.6 Data Input/Output

Matlab provides many functions that handle file I/O operations for different data types, including audio and image files. These built-in functions have been utilised in the software to allow import and export of data from and to external files. It is also possible to store and retrieve program variables from disk using the MAT-file format, which enables the signal data to be easily stored with any required settings parameters.

3.6.1 Audio Import/Export

The `wavread` and `wavwrite` functions allow audio data to be stored in the Microsoft WAVE (.wav) sound file format, which forms the basis of importing and exporting audio in the 2D Fourier software tool. The function `import_audio` was written to allow audio data to be imported and it is called when the user selects 'Import Audio' from the 'File' menu within the main GUI window. Listing 3.2 shows an abbreviated version of the `import_audio` function. The function brings up a file chooser GUI that will return the file name and path of the audio file to be imported. Matlab provides the `uigetfile` function to do this and also a `uiputfile` for saving to a file. The structure in `import_audio`, to open a file browser and then proceed if a file is chosen, is used for all data I/O operations in the software tool.

The `wavread` function is then used to get the audio data and also the sample rate and bit depth properties. Also the function checks if the audio data is stereophonic and if so converts it to mono. This is because the 2D Fourier tool currently performs all analysis and processing only on monophonic signal data. Performing the software investigation for

stereophonic data would have been much more difficult, it would be simpler to redevelop it now the investigation has been carried out.

```
function import_audio(hObject, handles)
%LOAD_AUDIO reads the desired .WAV file and stores it in the data
    structure

    %   use dialog to choose the audio file
    [handles.source.file,handles.source.audio_path,FilterIndex] = ...
        uigetfile('*.wav','Select a WAV file',...
        '/Users/chris/Documents/MEng Project/Matlab/Media/Audio/');
    if not(handles.source.file==0)
        %   set the source indicator
        handles.source.type = 'audio';

        [handles.data.audio,handles.data.audio_settings.Fs,handles.data.
            audio_settings.nbits] = wavread([handles.source.audio_path,
            handles.source.file]);
        % if the audio data is stereo then convert to mono
        if(size(handles.data.audio,2)==2)
            handles.data.audio = mean(handles.data.audio,2);
        end
        ...
    end
end
```

Listing 3.2: Importing Audio Using `wavread`

The rest of the function, which was omitted, calculates some properties of the audio data and then sets the variables required to allow processing operations to be applied, before calling the `analyse_audio` function to bring up the analysis settings menu.

When the 'Export Audio' option is chosen from the 'File' menu, the `export_audio` function is called, shown in listing 3.3. This is much simpler than the import operation. It brings up a file browser to select the save file and then, if a file is named, it calls a utility function that was written to remove the suffix from a file name. Then it uses the `wavwrite` function to save the audio data in a .wav file using the chosen file name and path. The GUI data is finally updated to retain the new `source` structure settings.

```
function export_audio(hObject,handles)
%SAVE_AUDIO will save the audio data as a WAVE file.
    [handles.source.file,handles.source.audio_path,FilterIndex] = ...
        uiputfile('*.wav','Save audio as a WAV file',...
        [handles.source.audio_path,handles.source.file,'.wav']);

    if not(handles.source.file==0)
        handles.source.file = remove_suffix(handles.source.file);
        wavwrite(handles.data.audio, handles.data.audio_settings.Fs, ...
            [handles.source.audio_path,handles.source.file,'.wav']);
        guidata(hObject, handles);
    end
```

Listing 3.3: Exporting Audio Using `wavwrite`

3.6.2 Image and Spectrum Export

As stated in section 3.2.1, the initial signal analysis steered the software towards using only audio source material so there is no import capability for the raster image or 2D spectrum data. It is however possible to export both displays as an image using the `imwrite` function. This export functionality is intended only to store the display of the data at screen resolution, it is not an accurate signal representation. The 2D data representations can sometimes have a height of only a few rows of data, which would not produce a displayable image. The software tool provides other means to store the data precisely for continued use in the software tool.

Matlab provides the function `getframe` to return a snapshot of a current figure or set of axes. The resulting pixel map can then be saved as an image. The function `save_2D` was written to allow the export of the raster image and 2D spectrum. It has a conditional argument to determine which display to save but it follows the same general structure as the `export_audio` function. The main difference is that an image has to be saved for each frame display within the signal, hence a loop is set up to display and capture each frame before redisplaying the original frame, indicated by the `cur_frame` variable. Listing 3.4 shows this display, capture and export loop for the raster image.

```
for frame = 1:nframes
    plot_image(handles.data.image{frame},...
        handles.data.audio_settings.nbits,...
        handles.axes_image,handles.source.type);
    I = getframe(handles.axes_image);
    imwrite(I.cdata, [path,handles.source.file,...
        '_',num2str(frame),'.tif'],'tiff');
end
plot_image(handles.data.image{cur_frame},...
    handles.data.audio_settings.nbits,handles.axes_image,...
    handles.source.type);
handles.source.image_path = path;
```

Listing 3.4: Exporting Raster Image Display

3.6.3 Program Data Saving and Loading

Matlab provides the built in functions `load` and `save` to enable storage of program variables on disk. This has been utilised in the software tool to allow the user to store the signal data, in all four representations along with its accompanying analysis parameters. The software can therefore load a signal and immediately enter a working state where the data is able to be analysed and processed.

The data is stored in a MAT-file, however the default suffix (.mat) can be changed, any file can be treated as a MAT-file when opened with the `load` function. A suffix was chosen to identify files specific to the software tool, .tda (two-dimensional audio) is used and the functions `load_tda` and `save_tda` have been written to allow these files to be used. These functions are very simple; a file browser dialogue is presented as with the other I/O functions introduced in this section and then if a file is chosen, the data is stored/retrieved.

There are three data structures stored in each TDA-file, `data`, `original_data` and `processes`, which are described in table 3.1. All the data representations and analysis settings are completely contained within `data` but the other two structures are required in order to retain the 2D Fourier processing chain and its settings, this is covered in chapter 5. In order to save the structures they have to be renamed so that they are not within a higher level structure, therefore when loading the variables have to be reinserted into the `handles`

structure. Once a TDA-file has been loaded, the `loaded` function is called which sets up the GUI and internal variables accordingly.

3.7 Audio Player

The audio player allows the audio data to be played through the host computers audio output device at the sample rate stored in the audio settings, which is obtained from the source audio file. The software tool uses Matlab's `audioplayer` object to control audio playback. This object is created by passing the audio data and settings into the `audioplayer` function, as shown in listing 3.5. Once instantiated the `audioplayer` object provides methods and properties that allow programmatic control of its functionality.

```
handles.player = audioplayer(handles.data.audio,handles.data.
    audio_settings.Fs, handles.data.audio_settings.nbits);
```

Listing 3.5: Creation of the Audioplayer Object

The functionality of the software tool's audio player is relatively simple. It is controlled using the push button interface shown in figure 3.5. The 'Play' and 'Stop' buttons initiate and terminate the audio playback using the `audioplayer` object's `play` and `stop` functions. There are two additional playback options, improving the value of the audio player as an analysis tool. The 'Loop' button switches continuous looping of the audio on or off. The other button toggles between 'All' and 'Frame', and it determines whether the whole audio signal is played or just the current frame of audio (for more detail on frames see chapter 4).

Figure 3.5: Audio Player User Interface

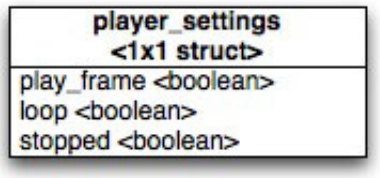


Figure 3.6: Data Structure for Audio Player Settings

The audio player's settings are stored in the `player_settings` data structure, shown in figure 3.6. The variable `play_frame` is set to true when only the current frame should be played. Playing a subsection of the audio signal is quite simple with the `audioplayer` object. The start and end sample indices are passed in as an argument to the `play` function. The sample indices for the current frame are stored in the `frame_settings` structure. The audio player's 'Play' push button has a callback function that calls the `play_audio` shown in listing 3.6.

```
function play_audio(handles)
%-- Plays the audio data using the audioplayer object. The 'play_frame'
%setting determines between playing a subsection of audio or the
%whole array.
    if handles.player_settings.play_frame
        play(handles.player,[handles.frame_settings.frame_start handles.
            frame_settings.frame_finish]);
    else
        play(handles.player);
    end
```

Listing 3.6: Playing Audio

The `loop` variable corresponds to the state of the 'Loop' toggle button on the audio player's user interface. If `loop` is true then the audio should resume playing from the beginning unless the 'Stop' button has been pressed. The `audioplayer` object's `StopFcn` property can be set to define a function that is called each time the `audioplayer` stops playback, whether it has played to the end of the audio or been stopped using `stop`. In order to differentiate between these two situations, the `stopped` variable is used. Pressing the 'Play' button sets `stopped` to false and it only becomes true when the 'Stop' button is pressed.

Listing 3.7 shows how the `StopFcn` property is set to call the function `stopFcn` and how this function controls looping.

```
set(handles.player,'StopFcn',{@stopFcn,handles});

function stopFcn(hObject,eventdata,handles)
%-- stopFcn is called when the audioplayer stops.
    % restart playback if loop is true and player stopped naturally
    if ~handles.player_settings.stopped && handles.player_settings.loop
        play_audio(handles);
    end
end
```

Listing 3.7: Player Looping Using StopFcn

Chapter 4

Two-Dimensional Audio Analysis

A large proportion of the project was devoted to investigating 2D audio analysis techniques and developing the software capabilities to employ these techniques appropriately and efficiently. This chapter describes the details of 2D audio analysis as performed in this project, both from theoretical and implementation perspectives. It has already been established in section 3.1.1 that two 2D signal representations are required, one in the time domain and one in the frequency domain. The properties of these signal representations will be fully explained. However it is first necessary to explain the analysis options presented to the user upon importing audio into the software tool, since this defines the structure of the signal representations. The chapter will also detail the implementation of analysis tools within the software.

4.1 Analysis Process

Once the audio data has been imported into the software tool, a series of processes must be carried out to obtain and display the various data representations. With regards to the 2D audio representations this requires specification of the initial analysis requirements using the GUI described in section 4.5. This is done using the `analysis_settings` function, which stores the chosen parameters in the `analysis_settings` data structure.

The function `analyse_audio` was written to carry out the initial signal analysis once the settings have been determined. It initialises the cell array variables `image` and `FT` at the

required size as well as the necessary variables within `analysis` structure, which vary according to the chosen settings. The `calc_image` function is called to convert the audio to one or more images by rasterisation, according to the analysis settings and then the `spec2D_init` function obtains the complex Fourier data from the image. Once both 2D representations have been obtained, the `loaded` function is called, as with TDA-files, to adjust the settings of the GUI and its components to allow the user access to analysis and processing functionality. The `loaded` function ends by calling `display_data` which will plot the data representations on the GUI. The details of plotting the raster image and the 2D Fourier spectrum will be covered in sections 4.2.5 and 4.4 respectively. The `display_data` function ends by storing the `handles` structure as GUI data for the main figure window, to retain the new signal data and settings.

4.1.1 1D Fourier Spectrum

The 1D Fourier data is calculated as it is plotted since it is a quick straightforward process and the data doesn't need to be stored for processing purposes. The 1D Fourier spectrum can either be for the whole audio array or the current frame of data represented by the displayed 2D images, this option is available from the 'Plot Settings' menu. Other options have been made available to provide a useful analysis plot. The user can select to view either the magnitude or phase spectrum and set either axis to linear or logarithmic scaling.

4.1.2 Automatic Normalisation of The Signal

The `normalise` function is called at the start of `display_data`. This function was written to scale the time domain data representations so that their maximum absolute value is one. This provides an appropriate display level and audible playback level in case a signal is too low/high in amplitude. It is an optional process, determined by the `auto_norm` variable in the `opt` structure, and it can be adjusted from the 'Options' menu. This option is most useful when 2D Fourier transformations have been applied to the signal that might severely affect its amplitude.

4.2 Raster Scanning

Raster scanning was introduced in section 2.5.1 as a simple method of converting between 1D and 2D data representations with a one-to-one sample mapping. The process of rasterisation serves as a gateway to 2D audio analysis, converting from the audio signal to one or more raster images.

4.2.1 Raster Image

The raster image is a 2D time domain representation of audio with two temporal axes. The horizontal axis has the time granularity of an individual sample period, that is $1/f_s$ seconds, and the vertical axis has a lower time granularity dictated by the width of the image, which can be defined as w/f_s seconds, where w is the image width in pixels. The amplitude of the audio signal at each sample point is displayed as a grayscale value creating an image display of the time domain signal.

4.2.2 Implementation of Rasterisation

The rasterisation process has been implemented in Matlab in `rasterisation`, a low-level function that has been abstracted from its application. It has two input arguments, the one-dimensional data array to be rasterised (`array`) and the integer width for the resulting image (`width`).

The function was designed with the option of row overlap in mind, which would abandon one-to-one mapping in favour of separating vertical time granularity from row width. This feature was not required in the final implementation hence the `hop` variable is set to equal `width` at the start of the function. To ensure fast operation it was decided that the `width` variable would always be integer valued and any variable passed in as `width` by the software tool would always be an integer. The `rasterisation` function is shown in listing 4.1.

```
function image = rasterise(array, width)
%RASTERISE converts between data from a 1D representation to a 2D
%representation using raster scanning.
%
```

```
%This function can be used to convert from a monophonic audio signal to a
%grayscale image.
%
%array = the 1D data representation
%width = the required width of the 2D output
%
%image = the 2D data representation

    hop = width;
    %calculate the required image size
    height = ceil(length(array)/hop);
    %create an empty matrix
    image = zeros(height,width);
    %rasterise data
    i=1;
    % add in this line for when image has only 1 row
    if length(array)>hop
        for arrayindex=0:hop:length(array)-width-1
            image(i,:) = array(arrayindex+1:arrayindex+width);
            i = i+1;
        end
        arrayindex = arrayindex+hop;
        rem = length(array)-arrayindex;
        image(i,1:rem) = array(arrayindex+1:length(array));
    else
        image(1,1:length(array)) = array;
    end
```

Listing 4.1: Rasterisation Process

The last row of the image must be assigned separately since it is likely that it will not be entirely filled by the remainder of the 1D array. If this is the case, the unassigned samples will all be zero valued and appear grey, since the image data has the same value range as the input data; this is equivalent to adding silence to the end of an audio signal.

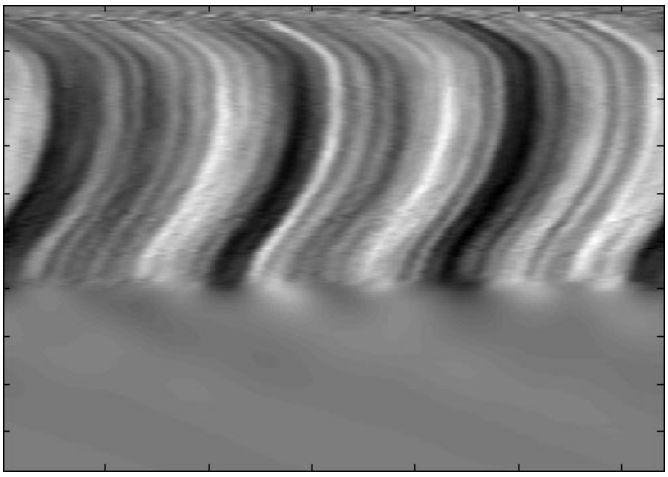
4.2.3 Raster Image Width

A raster image is most useful when the width of the image corresponds to a periodicity within the audio signal. The variation of this periodicity over time can then be observed much more easily than using an audio waveform display. In [39] the fundamental period of an audio signal was used to obtain a raster image of audio signals, however during this investigation it has been established that any periodicity within the signal will provide a useful raster image. This period could be harmonically related to the fundamental pitch period or alternatively it could be larger, corresponding to a rhythmic periodicity in an audio signal. Figure 4.1 shows two raster images, one of a cello playing a C# at approximately 69 Hz with the width corresponding to the fundamental period and the other of an electronic drum beat at a tempo of 120 bpm (beats per minute) with the width corresponding to one crotchet/quarter-note beat.

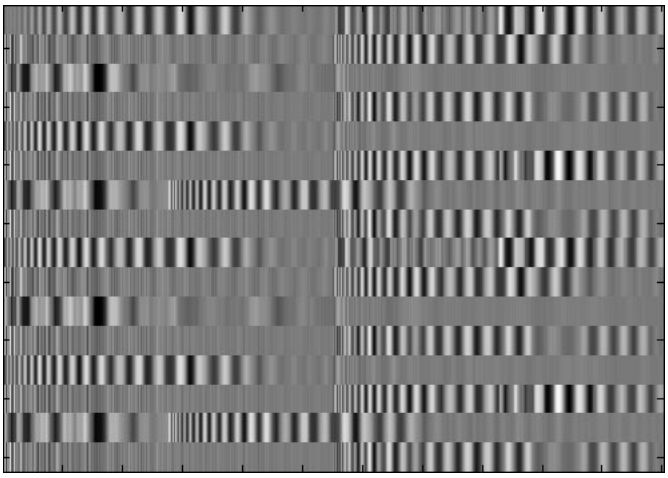
Correct assignment of the raster image width is important during 2D Fourier analysis. The 2D Fourier transform analyses the sub-sonic frequency variation in the audible frequency bins of each row, see section 2.2.7. If the image dimensions correspond to a periodic element of the signal then the Fourier data representation will be clearer. These details are fully explained in section 4.3.

4.2.4 Limitations of Raster Image Width

By the current definition each pixel in the raster image corresponds to a single audio sample, but it is rare for a pitch to correspond to an integer sample period and often a rhythmic duration such as a single beat will not be represented by an integer number of samples either. However an image file or data matrix must have integer dimension sizes in terms of pixel/sample units so the periodicity within the signal cannot be represented precisely.



(a) Cello C# at $\sim$ 69 Hz - Fundamental Period - Width 640 samples



(b) Electronic Drum Beat 120 bpm - Quarter-note Beat - Width 22050 samples

Figure 4.1: Raster Images Demonstrating Different Periodicities In Audio Signals

The idea of using interpolation to obtain an integer period, adjusting the sample rate of the data, was considered. However this was deemed an unnecessary extension of the program requirements, because it would have added a lot of computing overheads. More importantly, it was decided that at this stage in investigating the properties of 2D audio analysis, the signal periodicity could be appropriately approximated using only integer periods since it was more important to investigate the potential of this signal processing paradigm and understand its properties than to spend a lot of time fine tuning individual

processes. It only becomes a large problem at periods of less than 50 samples, where the jump in pitch for a width change is > 20 Hz. The corresponding frequencies are very high, approaching the limits of human hearing and it is much less common for musical notes to be in this frequency range. It would not be difficult to incorporate sample rate changes into the software to allow more accurate 2D analysis, now that the investigation has been carried out and the software developed.

4.2.5 Displaying The Raster Image

The raster image data is displayed using the `image` function, provided by Matlab to display image data on a set of plot axes. This function can either accept true RGB colour data or indexed data in the range [0 1] that uses a colour map to obtain the colour value. Since the raster image is grayscale, it has to be scaled to the indexed data range and then the Matlab `colormap` function can be used to select the `gray` map. Listing 4.2 shows the plot_image function used to display the image data on the axes.

The property `DefaultImageCreateFcn` is set to `'axis normal'` which ensures that the image is displayed at the axes dimensions rather than the image dimensions. The image can often be much larger in one dimension than the other, so that if it was displayed with square pixels it would be very difficult to display within a GUI. It was decided that this display method is more useful to the user. The `plot_image` function also labels the plot and its axes.

```
function plot_image(image_data,bits,plot_axes,source)
%PLOT_IMAGE plots a gray-scale image on the given axes at the correct bit
%depth.

    % convert image_data from range [-1 1] to [0 1]
    image_data = (image_data+1)/2;
    %  select axes
    axes(plot_axes);
    %  ensure pixels are displayed square
    set(0,'DefaultImageCreateFcn','axis normal');
    %  display image
    image(image_data);
```

```
% label the plot
title('Raster Image');
set(get(gca,'XLabel'),'String','Pixels');
set(get(gca,'YLabel'),'String','Pixels');
colormap(gray);
```

Listing 4.2: Plotting The Raster Image

4.3 Two-Dimensional Fourier Transform

The 2D Fourier transform can be used to provide a new perspective on frequency domain audio analysis, by simultaneously giving the audible frequency content and the sub-sonic rhythmic variation in audible frequency, as described in section 2.2.7. It is the core of this software investigation and the aim was to understand its properties for audio analysis. The 2D FFT was used in the software tool to obtain the 2D Fourier spectrum of audio data, which could then be displayed to the user.

4.3.1 Obtaining the 2D Fourier Spectrum in Matlab

As described in section 2.1.2, Matlab provides the function `fft2` to perform the 2D discrete Fourier transform, which returns the Fourier transform data matrix. The function `spec2D_init` was written to obtain the complex Fourier data from each raster image. The Matlab functions `abs` and `angle` can be used to obtain magnitude and phase components from the complex matrix but this is not done immediately, the Fourier transform data is stored in the `FT` cell array within the `data` structure.

4.3.2 Analysis of the 2D Fourier Transform

In order to understand the 2D Fourier domain representation of a signal, it is important to analyse the process by which this representation was obtained. The 2D Fourier transform can be divided into two separate stages. First the 1D DFT is computed for each column, resulting in a complex valued 2D array separately describing the frequency content of each column of original data. Secondly the 1D DFT is computed for each row of this

intermediate complex array, yielding the resulting 2D Fourier data array. The result is identical if the rows are computed in the first stage and the columns second, and in terms of our analysis this is the most logical order. The intermediate data can then be viewed as a series of STFT frames with no overlap and a rectangular window. Each column of the data corresponds to the frequency component given by equation 2.3 where N is the width of the 2D array and v is the column index.

The second stage of the 2D Fourier transform requires a DFT of complex data, which has slightly different properties to the real DFT [31]. The periodicity of an N-point real DFT means that the frequency spectrum is reflected about $\frac{N}{2}+1$ (section 2.1.1), and half the data is redundant. Whereas the complex DFT has no redundancy and all N points are required to fully represent the signal. In other terms, both the positive and negative frequencies are required with the complex DFT. However of the four quadrants of the 2D Fourier data, two will still be redundant due to the periodicity of the DFT in the first stage. It was decided to show all four quadrants of the 2D Fourier spectrum in the display, as is the convention in image processing literature [16, 31], since it allows the user to observe the relationship with the raster image more intuitively.

The most important aspect of this investigation was to determine what information can be taken from the 2D Fourier spectrum of an audio signal. To understand the representation of an arbitrary signal in this 2D spectrum, simple signals must first be analysed so that the basic 2D Fourier component can be defined. The concept of a 2D sinusoid, with a direction parameter, was introduced in section 2.3.1. Any single point on the 2D Fourier spectrum grid represents a sinusoidal signal travelling in a certain direction within the raster image. However in [23], Penrose describes the 2D spectrum in terms of audible frequencies that vary at a lower rhythmic frequency.

As stated in section 2.1.3, the 2D discrete Fourier transform is periodic, an $M \times N$ spectrum of an image has a period of M samples in one dimension and N samples in the other. Each point has a matching point with opposite phase due to aliasing of a sampled signal, except (0,0), ($M/2$,$N/2$), (0,$N/2$) and ($M/2$,0) which cannot be represented by any other point on the spectrum. In a spectrum with two odd dimensions, the only point that does not have an equivalent is (0,0) since the other co-ordinates do not exist. This is the DC content of the signal, so if an audio signal has a constant non-zero value, (0,0) will be the only point in the 2D Fourier spectrum with a non-zero magnitude value and no point will have

non-zero phase.

Signal Components With Audible Frequency Only

A sinusoidal signal with an integer period size can have a raster image that contains exactly one period in each row. When this is analysed using the 2D Fourier transform, the resulting spectrum contains only two points both having zero rhythmic frequency since there is no vertical variation in the raster image, and with the symmetrical positive and negative audible frequency of the sinewave, as shown in figure 4.2. The 2D spectrum has been zoomed in in this figure, to more clearly display the two points.

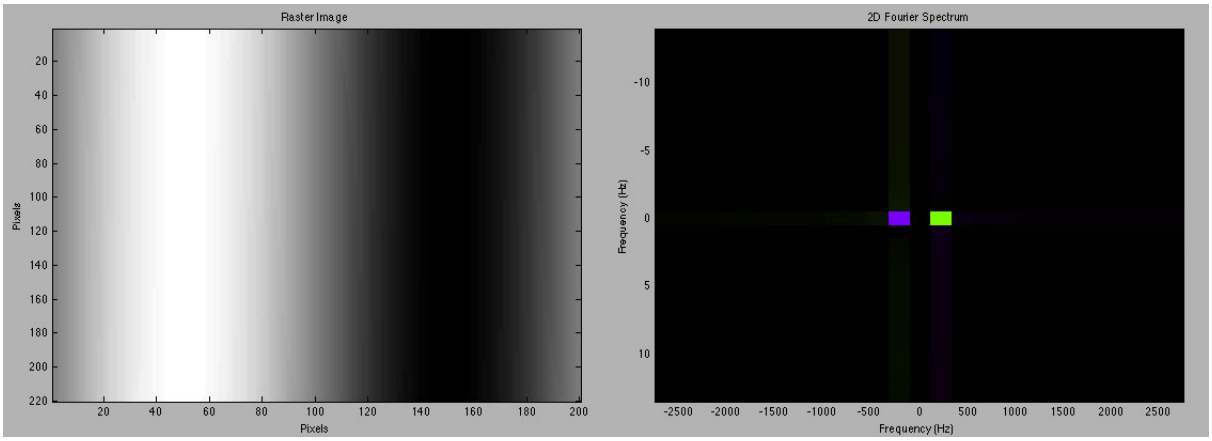


Figure 4.2: 2D Analysis of a Sinusoid With Correct Raster Width

Signal Components With Rhythmic Frequency Only

When there is no variation across each raster image row but a clear periodicity in the columns, the opposite occurs. The 2D Fourier spectrum demonstrates two points with zero audible frequency and symmetrical positive and negative rhythmic frequency, as shown in figure 4.3.

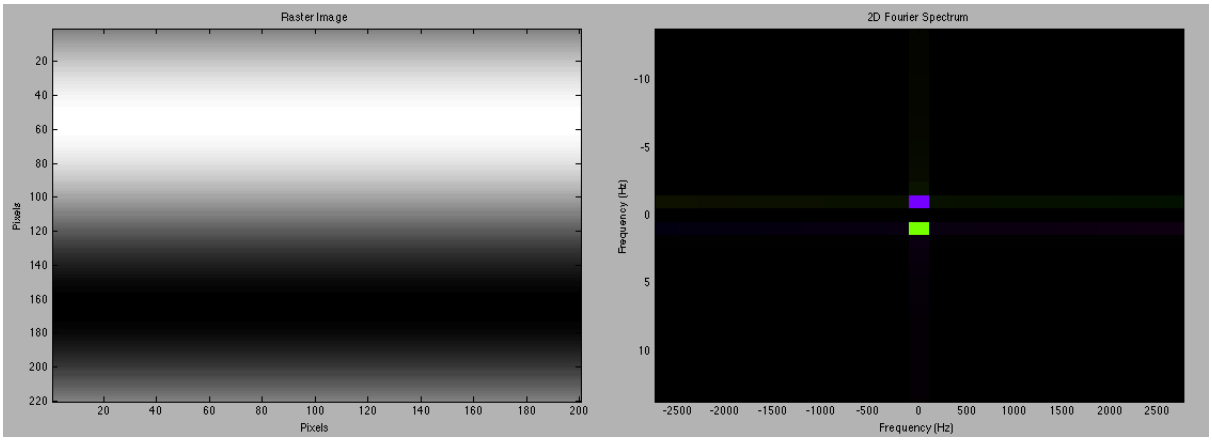


Figure 4.3: 2D Analysis of a Rhythmic Sinusoid With Correct Raster Height

These two initial examples have both had a frequency component in one of the two axes and zero frequency in the other, resulting in two points of equal magnitude and opposite phase, on opposite sides of the (0,0) centre point in the respective frequency axis.

Signal Components With Audible & Rhythmic Frequency

A signal with a defined frequency in both axes is an amplitude modulated sinusoid. Amplitude modulation is conventionally considered in terms of a carrier frequency and a modulation frequency. In the terms used in this project, carrier frequency is synonymous with audible frequency whilst modulation frequency is synonymous with rhythmic frequency.

1D Fourier analysis defines amplitude modulation using two sinusoids with stationary amplitude characteristics, f_1 and f_2. The mean of these two frequencies gives the carrier frequency whilst their difference gives modulation frequency. In 2D Fourier analysis amplitude modulation is given by a single sinusoidal component which has an audible frequency, f_a, and a rhythmic frequency, f_r. The relationship between 1D and 2D Fourier analysis of an amplitude modulated sinusoid is shown by the following equations:

$$f_1 = f_a - \frac{f_r}{2} \tag{4.1}$$

$$f_2 = f_a + \frac{f_r}{2} \tag{4.2}$$

The AM sine wave can be considered as the two sinusoids f_1 and f_2 which have opposing angles within the raster image.

If the signal's carrier frequency has a period of exactly the raster image width and the modulation frequency has a period corresponding exactly to the image height, a well defined 2D Fourier representation is obtained, where the effects of rectangular windowing are avoided in both dimensions. The 2D Fourier spectrum of this signal has four points, as shown in figure 4.4. It appears that there are two lines of symmetry within the spectrum, however we know that only half of the spectrum contains redundant information.

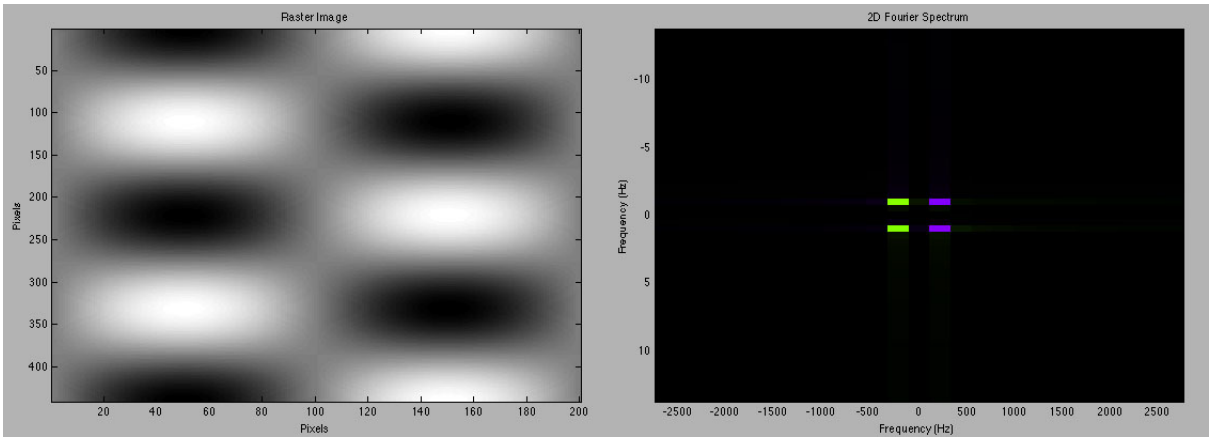


Figure 4.4: 2D Analysis of an AM Sinusoid

On closer inspection the data only follows the expected symmetry of the 2D Fourier transform, as given by equations 2.6 and 2.7, with two pairs of points matching in magnitude and opposite in phase in diagonally opposing quadrants. However the position of these points is symmetrical and the values are similar, with a magnitude difference of 0.19% and a phase difference of 1.76% from the largest value to the smallest.

Analysis of The Complex DFT

The complex DFT needs to be better understood to explain the 2D spectrum of this signal, since it appears that there is some form of aliasing occuring in the second stage of the 2D Fourier transform even though the data is not symmetrical.

The linearity of the DFT [16] is given by the following equation:

$$af_1(x,y) + bf_2(x,y) \Leftrightarrow aF_1(u,v) + bF_2(u,v) \tag{4.3}$$

Therefore summing two signals in the frequency domain is equivalent to summing them in the time domain. This is demonstrated in figure 4.5 which shows the 2D Fourier analysis of the sinusoid of figure 4.2 summed with the amplitude modulated sinusoid of figure 4.4 in the time domain.

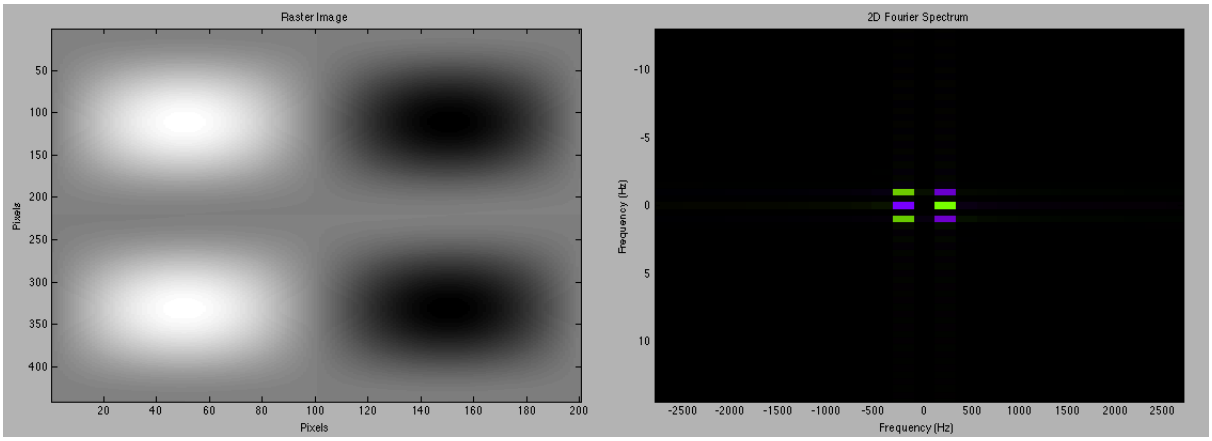


Figure 4.5: Demonstrating The Linearity of the 2D DFT

By viewing a complex signal as two separate signals, the real and imaginary components, the complex DFT might be analysed and the 2D spectrum understood. If two real input signals were used, the Fourier transform would produce a complex output for each of the signals, demonstrating aliasing and hence having symmetry about zero frequency. The Fourier domain data could then be summed to get the spectrum of the two signals combined.

The complex input signal, $f(x)$, can be viewed as the sum of two real signals, with one multiplied by j:

$$f(x) = r(x) + ji(x) \quad \text{where} \quad \begin{aligned} r(x) &= \mathbb{R}\left(f(x)\right), \\ i(x) &= \mathbb{I}\left(f(x)\right), \\ j &= \sqrt{-1} \end{aligned} \tag{4.4}$$

The symbols $\mathbb{R}$ and $\mathbb{I}$ represent the real and imaginary components of a signal respectively. The Fourier transform of a complex input can then be broken into the sum of two complex signals, also with one multiplied by j.

$$r(x) + ji(x) \Leftrightarrow R(x) + jI(x) = F(x) \tag{4.5}$$

When the two resulting complex signals are also broken into two real components, the structure of the overall Fourier domain data can be observed, shown in equation 4.8.

$$R(x) = R_r(x) + jI_r(x) \quad \text{where} \quad \begin{aligned} R_r(x) &= \mathbb{R}\left(R(x)\right), \\ I_r(x) &= \mathbb{I}\left(I(x)\right) \end{aligned} \tag{4.6}$$

$$I(x) = R_i(x) + jI_i(x) \quad \text{where} \quad \begin{aligned} R_i(x) &= \mathbb{R}\left(I(x)\right), \\ I_i(x) &= \mathbb{I}\left(I(x)\right) \end{aligned} \tag{4.7}$$

$$F(x) = R_r(x) - I_i(x) + j\left(I_r(x) + R_i(x)\right) \tag{4.8}$$

Therefore the DFT of complex time-domain data results in complex frequency-domain data, where:

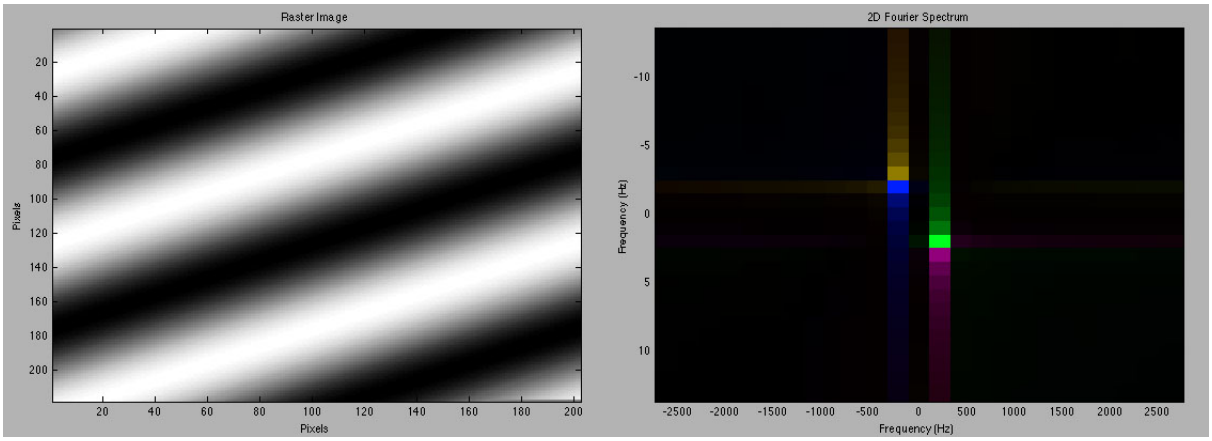
- The real component of the frequency-domain data is composed of the real-component of the DFT of the real time-domain data minus the magnitude-component of the DFT of the imaginary time-domain data.
- The imaginary component of the frequency-domain data is composed of the imaginary-component of the DFT of the real time-domain data plus the real-component of the DFT of the imaginary time-domain data.

This process would need further analysis to investigate the numerical relationship between positive and negative frequencies of the complex DFT, but it demonstrates how the 2D spectrum can have four symmetrically placed points due to the periodicity of the separate real and imaginary components of the complex signal that is output by the first stage of the 2D FFT and input into the second stage. The periodicity of the real DFT in the first stage of the 2D FFT means that the four symmetrical points can be divided into two pairs. Each pair of points has an audible and rhythmic frequency of opposite sign, identical magnitude and opposite phase.

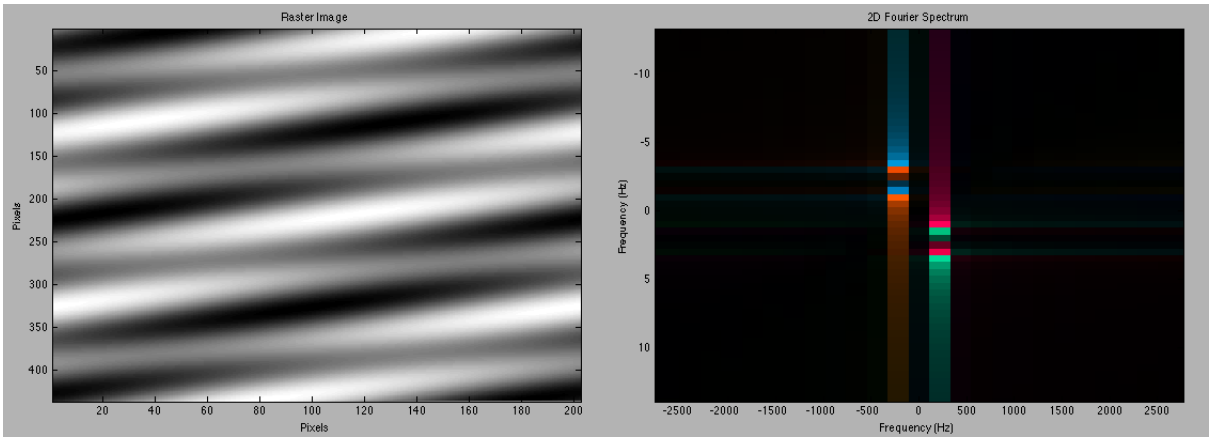
It has been established that when the raster image size accurately depicts horizontal and vertical periodicities, the 2D Fourier spectrum displays four clear points $(f_r, f_a), (-f_r, f_a), (f_r, -f_a)$ and $(-f_r, -f_a)$. This signal is an amplitude-modulated sinusoid where f_r is the modulation (rhythmic) frequency and f_a is the carrier (audible) frequency of the signal.

Signal Components Not Aligned With The Raster Image Dimensions

In a more complex audio signal there will be frequency components that don't fit the raster image and therefore run at an angle such as the 2D sinusoids shown in figure 2.4. Signal components such as this will have non-stationary phase in both frequency axes leading to spectral smearing as shown in figure 4.6a.



(a) Sinusoid With Incorrect Raster Image Width



(b) AM Sinusoid With Incorrect Raster Image Width

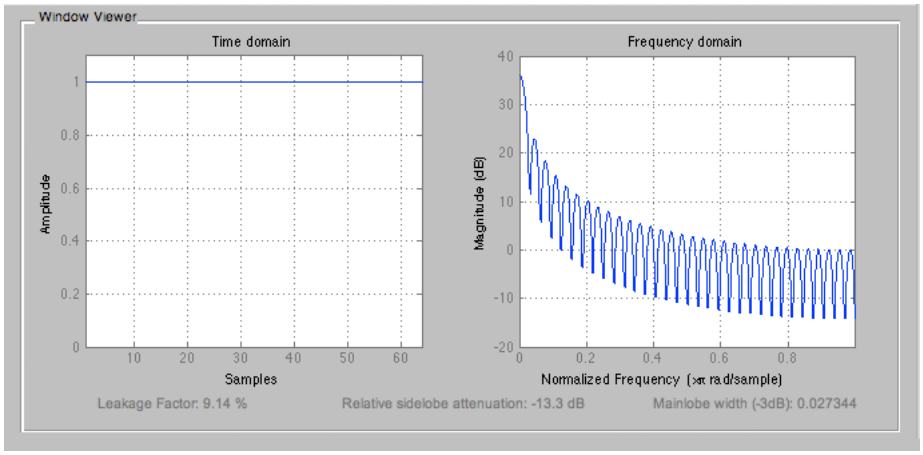
Figure 4.6: Spectral Smearing In Signal Components Not Aligned To Raster Dimensions

An AM sinusoid with the incorrect raster width demonstrates a similar spectral smearing with its 4 points skewed to create a line perpendicular to the direction of the wave (figure 4.6b).

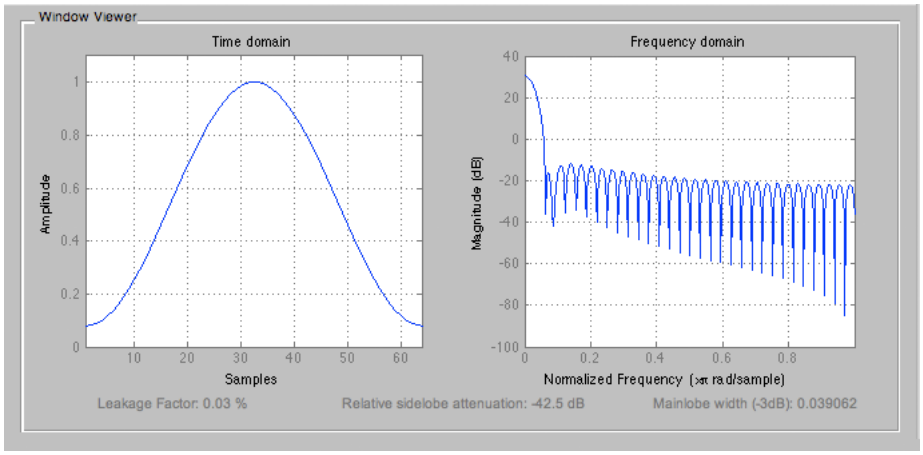
Effects of Rectangular Windowing

The rasterisation process applies a rectangular window function to the audio signal to extract each row of the raster image. When signal components are correctly pitch-synchronised the audible frequency analysis will display these components by a single point. Likewise in the vertical axis of a raster image, when a sub-sonic signal component has a precise period synchronisation with the image dimensions it can be represented by a single point in the rhythmic axis. However when there is no synchronisation the effects of the rectangular window distort the 2D Fourier domain signal representation.

Each non-synchronised 2D spectral component is represented by its actual audible and rhythmic frequency convolved with the frequency response of the rectangular window in both axes. If a component is synchronised in one axis then the other axis will still show distortion as a result of this convolution.



(a) Rectangular Window



(b) Hamming Window

Figure 4.7: Window Function Time and Frequency Representations

The Matlab Signal Processing toolbox provides a window analysis tool which was used to observe the frequency response of a rectangular window, as shown in figure 4.7a. Figure 4.7b shows a Hamming window in the time and frequency domains to serve as a comparison. Both window functions have a frequency domain form that resembles a sinc function, with a main frequency lobe with many lower magnitude side lobes. The rectangular window has a narrower main lobe but at the expense of much higher magnitude side lobes. These side lobes cause the spectral representation of a non-synchronised sinusoidal component to leak energy into adjacent frequency bands, reducing the clarity of its representation.

4.3.3 Signal Analysis Using The 2D Fourier Spectrum

The investigation of section 4.3.2 allows a summary of features that can be observed in the 2D Fourier spectrum. The fundamental component of the 2D Fourier spectrum is a sinusoid modulated in amplitude by a lower frequency sinusoid, apart from any point that is 0 Hz in either frequency dimension, which is just a sinusoid.

Any sinusoidal components which appear identical in every row of the raster image must be harmonically related to the period of the raster image width and have no amplitude modulation. These components will be displayed on the 2D spectrum with a rhythmic frequency of 0 Hz and their precise audible frequency (both positive and negative), which is a multiple of the frequency:

$$f_{a0} = F_s/N \tag{4.9}$$

Where F_s is the sample rate, N is the width of the raster image plus padding and f_0 is the fundamental audible frequency of the analysis.

Any components which appear identical in every column but vary in each row are harmonically related to the fundamental rhythmic frequency of the analysis, which is given by:

$$f_{r0} = f_{a0}/M \tag{4.10}$$

Where M is the raster image height plus padding. These components will be displayed on the 2D spectrum with an audible frequency of 0 Hz and their precise rhythmic frequency, both positive and negative.

A signal component that is exactly periodic in both dimensions of the raster image is a sinusoid harmonically related to f_{a0} modulated by a sinusoid harmonically related to f_{r0}. It will be displayed as four points on the 2D spectrum, corresponding precisely to the positive and negative rhythmic and audible frequencies.

Any signal component that cannot fit a whole number of periods into a raster image row, will be displayed at an angle and the points of its frequency spectrum will be skewed so they are perpendicular to the direction of the sinusoid. This occurs with sinusoids with an amplitude modulation as well as those with no modulation. The energy of these components will also be spread across adjacent frequency bins, since its frequency does not directly match the centre frequency of any point on the spectrum.

The 2D frequency spectrum represents sinusoidal components with sinusoidal amplitude modulation. When the frequency of either the carrier or the modulator exactly matches the centre frequency of one of the 2D spectrum points for its respective axis, the component will be precisely modelled by a single bin in that axis. If the frequency of either carrier or modulator doesn't match the centre frequency of a 2D spectrum point, then its energy will be spread across several bins along the respective axis. This energy smearing is the same as in 1D Fourier analysis and might be solved using 2D implementations of the techniques used to improve spectrum resolution in one dimension [12]. The centre frequency of the points on both frequency axes of the 2D spectrum is defined by the width of the raster image so an informed choice of raster image width has to be made based on signal analysis in order to obtain a clear 2D spectrum.

4.4 Two-Dimensional Spectrum Display

The 2D Fourier spectrum plot needed to represent the complex Fourier data matrix clearly, allowing the user to take in a lot of information at once. Fourier data is more easily understood in the polar representation which is standard in 1D Fourier analysis, so the magnitude and phase components are extracted. The function `calc_spec2D`, shown in listing 4.3 was written to calculate the spectrum display data, from the complex Fourier data stored in the `FT` cell array in the `data` structure. The Fourier transform data is first shifted using the built-in `fftshift` function to place the DC component in the centre of the matrix. Then the `angle` function returns the phase component and the `abs` function then returns the magnitudes. The polar representation is converted to an RGB colour matrix as described in section 4.4.1.

```
function ret = calc_spec2D(FT, brightness, contrast, mode, ret_mag)
%PLOT_SPEC2D calculates a 2D spectrum given the Fourier transform data.

    %   get the magnitude component (centred)
    mag = abs(fftshift(FT));
    %   normalise
    max_mag = max(max(mag));
    if ret_mag
        ret = max_mag;
```

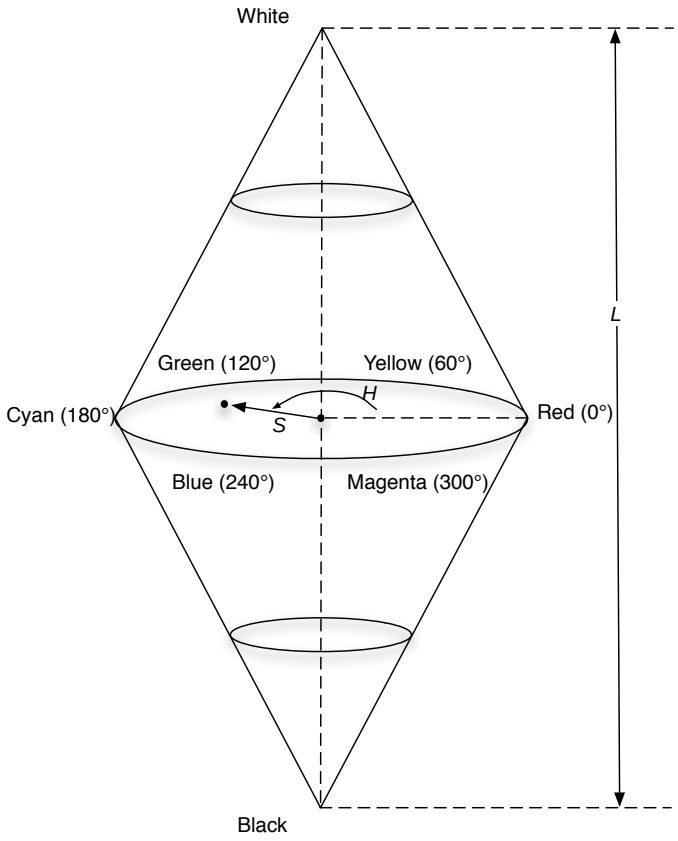
```
    else
        magN = mag/max_mag;
        %   add 1 before logarithm to prevent -inf value occuring
        magLN = log2(1+magN).^contrast;

        %   get the phase
        phase = angle(fftshift(FT));
        %   convert the polar representation to RGB colour representation
        spec2D = polar2colour(magLN, phase, brightness, mode);

        ret = spec2D;
    end
```

Listing 4.3: Calculating the 2D Fourier Spectrum Display Data

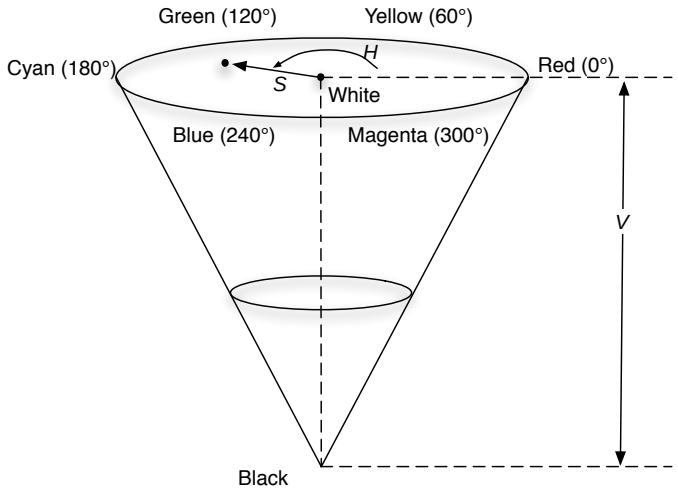
Figure 2.3 demonstrates that magnitude information gives a more intelligible visual display than the phase, allowing features of the raster image to be inferred [16]. Hence the first stage in development of the 2D Fourier spectrum plot was to display the magnitude spectrum as a grayscale image. The `image` function is used to display the Fourier spectrum on axes, as with the raster image, so the data must be in the range [0 1]. The magnitude data must therefore be normalised by dividing by the maximum magnitude value in the matrix. Also, by scaling the magnitude data logarithmically the visual display gives much more detail.

4.4.1 Colour Representation of Polar Data

The data for the `image` function has to be a three-dimensional (m-by-n-by-3) array of RGB colour values, for a grayscale display the three colour values should be equal. However the design of the 2D Fourier spectrum display was extended, based on the techniques in [15], to include phase information by converting from polar data to RGB colour data. The function `polar2colour` was written to perform this data conversion. It takes in the magnitude and phase data arrays of size (m-by-n) and returns a single (m-by-n-by-3) RGB colour array. This function was developed after correspondence with the author of [15] and with the benefit of the Java source code for that Java applet, both of which are available in appendix A.



(a) HSL Colour Space



(b) HSV Colour Space

Figure 4.8: Comparison of Colour Space Representations, after [1]

The initial conversion is based on the hue, saturation and lightness (HSL) colour model, shown in figure 4.8a. This system is often considered to be a far more intuitive representation of colour [35]. The phase component is mapped to hue and the magnitude to lightness, with a full saturation value. These mappings are demonstrated in the following equations, where FT is the complex Fourier data:

$$H = \frac{\angle FT + \pi}{2\pi} \tag{4.11}$$

$$L = \arctan(|FT|) * (2/\pi) \tag{4.12}$$

Matlab can deal with colour in the HSV colour space (figure 4.8b) but not HSL, so a conversion must be performed. The differences between HSV and HSL are given in [11] and are repeated here. Hue (H) is defined exactly the same in both systems. In the HSV system, value (V) and saturation (S_V) are defined as:

$$V \equiv \max(R, G, B) \tag{4.13}$$

$$S_V \equiv \frac{V - \min(R, G, B)}{V} \tag{4.14}$$

In the HSL system, lightness/brightness (L) and saturation (S_L) are defined as :

$$L \equiv \frac{\min(R, G, B) + \max(R, G, B)}{2} \tag{4.15}$$

$$\begin{aligned} S_L &\equiv \frac{\max(R, G, B) - \min(R, G, B)}{\max(R, G, B) + \min(R, G, B)} \qquad \text{if } L \leq 1/2, \\ &\equiv \frac{\max(R, G, B) - \min(R, G, B)}{2 - \max(R, G, B) - \min(R, G, B)} \quad \text{if } L > 1/2. \end{aligned} \tag{4.16}$$

Therefore to convert HSL to HSV, the V and S_V values need to be obtained from L and S_L as follows:

$$\begin{aligned} V &= L(1.0 + S_L) \qquad \text{if } L \leq 1/2, \\ &= L + S_L - L * S_L \quad \text{if } L > 1/2. \end{aligned} \tag{4.17}$$

$$S_V = (2 * (V - L))/V \tag{4.18}$$

When $S_V = 1$ the equations in the function `polar2colour` are valid, as shown in listing 4.4. The function finally calls Matlab's built-in `hsv2rgb` function, to convert the matrix from HSV colour mode to RGB. Each of the HSV components must be in the range [0 1],

however the lightness array is rescaled within `polar2colour` to prevent it from reaching 0. During testing it was found that a colour with a *value* component of 0 gives an RGB colour of [0 0 0], no matter what the hue is, hence losing the phase information.

```
% calculate lightness from mag
lightness = atan(mag/brightness)*(2/pi);
% adjust range to prevent 0 value
lightness = (lightness+0.001)/1.001;
% calculate saturation and value in range [0 1]
for m=1:size(lightness,1)
    for n=1:size(lightness,2)
        if(lightness(m,n)<=0.5)
            % saturation
            hsv(m,n,2)=1;
            % value
            hsv(m,n,3) = 2*lightness(m,n);
        else %lightness>0.5
            % saturation
            hsv(m,n,2) = 2-2*lightness(m,n);
            % value
            hsv(m,n,3) = 1;
        end
    end
end
% hue with no banding correction
hsv(:,:,1) = (phase + pi)/(2*pi);
```

Listing 4.4: HSL to HSV Conversion in the `polar2colour` Function

4.4.2 Spectrum Display Options

The function `polar2colour` has a `mode` input argument, which determines whether the RGB image returned displays the magnitude data, the phase data or both combined. In combined mode, the default, the data is calculated as in listing 4.4. For a magnitude only display, all of the RGB values are set to the normalised magnitude value, producing a grayscale image. To display only the phase information, saturation and value are both set to 1 and the hue shows the phase information. These options are presented to the user

through the 2D spectrum mode menu, a sub-menu of 'Plot Settings' and the selection is stored as a string in the `spec2D_mode` variable within the `plot_settings` structure.

4.4.3 Brightness & Contrast of Display

In order to allow a more flexible display, additional brightness and contrast parameters have been introduced into the algorithm and sliders are provided on the user interface to allow their adjustment. The brightness parameter adjusts the overall brightness of the display by scaling the magnitude values and the contrast parameter is used as an exponent of the magnitude data to adjust the display scale.

The brightness slider's value has a rangeof [0 0.995], and this is used in the following equation to obtain the `specBrightness` variable which has a range of [0.0314 1.633e+016] and is inversely proportional to the slider value. After some experimentation it was found that this range with a exponential style scale gave the most intuitive control settings.

$$\texttt{specBrightness} = -\tan\left((\text{slider\_value} - 1) * \pi/2\right) \tag{4.19}$$

This `specBrightness` variable is stored in the `plot_settings` structure, as is `specContrast`. It is used within the `polar2colour` function as a divisor to the magnitude component (normalised and log scaled) to scale it.

The contrast slider's value has a range of [-1 1] and this is converted to an exponentially scaled range of [0.2 2] using the following equation:

$$\texttt{specContrast} = 2 * 10^{(\text{slider\_value}-1)/2} \tag{4.20}$$

`specContrast` is used in the `calc_spec2D` function (listing 4.3) as an exponent of the logarithmic normalised magnitude component, which has a range of [0 1]. The plot in figure 4.9 attempts to visualise this display range adjustment. The normalised magnitude is equivalent to variable `magN` in the `calc_spec2D` function and the contrasted logarithmic normalised magnitude is equivalent to `magLN`. The given range of the `specContrast` variable was chosen to allow the most useful scale of contrast adjustment.

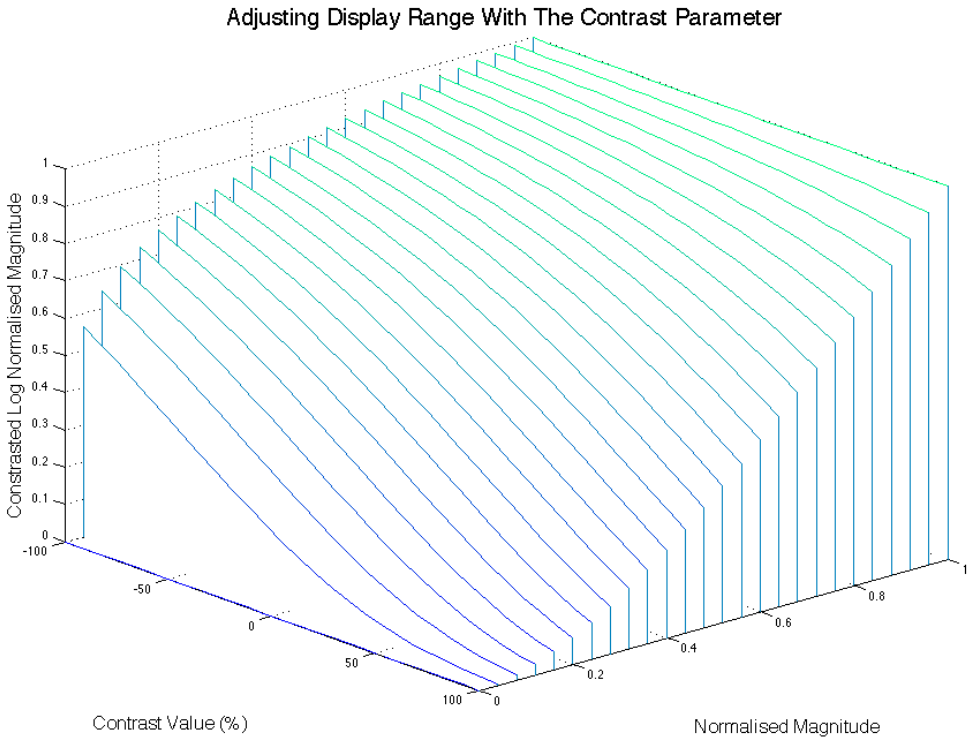


Figure 4.9: Display Range Adjustment Using The Contrast Parameter as an Exponent

4.4.4 Zero Padding

In [16] it is recommended that the image be zero padded so that the dimensions are both even numbers before performing the Fourier transform. This is to ensure that the same `fftshift` function can be used for the forward and inverse processes, with odd dimensions the forward and inverse processes are different. It is however more important for analysis puposes that there is a DC component at the centre of the spectrum, which can only be achieved with two odd dimensions. Therefore the raster images are padded with a single row and/or column of zeros, as necessary to obtain an odd numbered size, before the 2D FFT is performed in `spec2D_init`. This requires use of Matlab's `padarray` function, as shown in listing 4.5.

```
handles.data.spec2D_settings.height_pad(frame) = 1-mod(size(handles.data.
    image{frame},1),2);
```

```
handles.data.spec2D_settings.width_pad(frame) = 1-mod(size(handles.data.
    image{frame},2),2);
padded = padarray(handles.data.image{frame},...
    [handles.data.spec2D_settings.height_pad(frame) ...
    handles.data.spec2D_settings.width_pad(frame)],0,'post');
```

Listing 4.5: Pre-Padding The 2D Spectrum With `padarray`

This use of zero padding means that a reverse process for `fftshift` had to be written. Due to the specific requirements, the `rev_fftshift` function didn't need to work for even dimension sizes. It simply obtains the matrix size and moves the four quadrants as arrays into a new matrix of the same size, in their correct positions. Figure 4.10 demonstrates the process of shifting data quadrants for a matrix with odd dimensions.

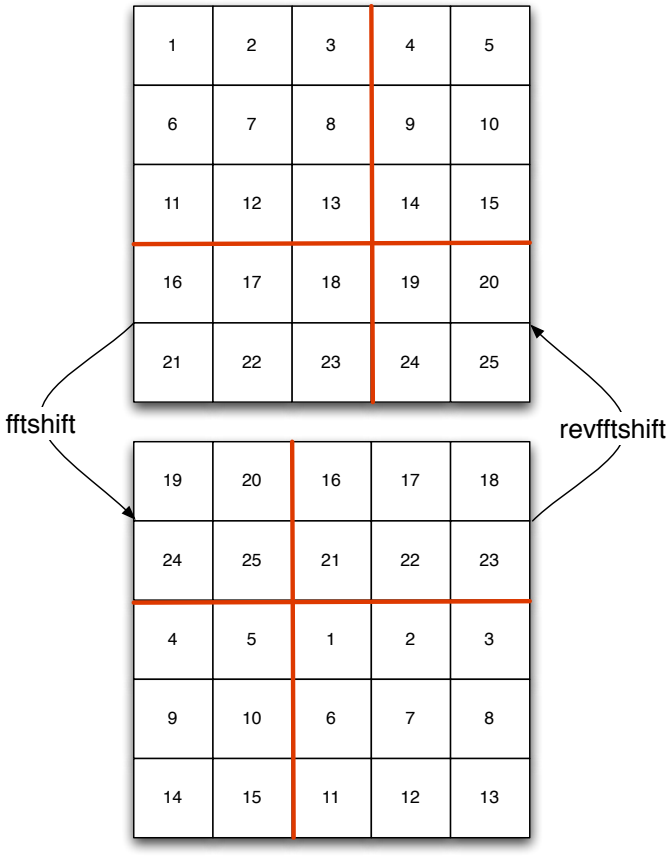


Figure 4.10: The Fourier Data Shift Functionality For Odd Dimensions

4.5 Audio Analysis Options

It is rare for a single analysis method to provide useful information about all types of audio signal; audio characteristics are hugely varied. An important part of this investigation was to determine suitable methods for analysing audio signals in two dimensions, and to understand what each method provides in terms of both signal analysis and subsequently possible signal transformation.

The 2D Fourier software tool provides the user with a GUI window when they import a new audio file or choose to reanalyse a current signal, this window is defined by the `analysis_settings` function. It presents a series of options that determine the way in which the audio signal is represented in two dimensions; these options were defined based upon an understanding of the underlying signal processing and the different types of audio signal the user may want to interact with.

4.5.1 Analysis Mode

There are two main modes of 2D signal analysis, *timbral* and *rhythmic*. They are classified by the factor that determines their raster image width. Timbral mode uses pitch detection (section 4.7.1) to determine the width of the image as the fundamental period size in samples, rounded to the nearest integer. Rhythmic mode uses the tempo and a note length, such as one quarter-note, to determine the width of the image in samples, again rounded to the nearest integer.

Each of these representations offers a different scale of rhythmic frequency. In timbral mode the upper rhythmic frequency boundary is in the order of tens and sometimes hundreds of Hertz, extending into the lower end of the audible frequency spectrum. Frequencies in this range describe the slow oscillations over a signal that are commonly related to the conceptual timbre of a sound, describing the evolving amplitude and audible frequency spectrum envelopes. In rhythmic mode the upper rhythmic frequency boundary is much lower, generally less than 10 Hz. This frequency range corresponds to conventional rhythmic frequencies, for example at a tempo of 120 bpm, eighth-notes occur at a frequency of 4 Hz.

The choice of raster image width affects the resulting 2D Fourier spectrum, as described

in section 4.3.3. Each analysis mode attempts to represent a different frequency axis accurately, setting the raster width accordingly. In rhythmic mode, the raster image width is set so that the frequency bins of the rhythmic axis are likely to correspond to prominent rhythmic components within the signal. The maximum frequency on the rhythmic axis is half the frequency of the beat duration that defines the raster image width. Timbral mode sets the raster image width so that the frequency bins of the audible axis correspond to harmonics of the fundamental pitch frequency.

4.5.2 Signal Frames

The initial analysis divides the audio signal into one or more frames, however the definition of a frame differs between the two modes. In timbral mode, the signal is divided into into sub-sections called frames before rasterisation. Each frame is subjected to 2D analysis and has its own raster image and 2D Fourier spectrum. The width of the raster image for each frame is determined by the calculated pitch of that signal sub-section. The reasoning for this methodology was that an audio signal could contain more than one note, and it might be necessary to divide the audio into several frames, each with a different width defined by its pitch, to get an accurate analysis.

In rhythmic mode the frame is synonymous with a row of the raster image and there is only one raster image and one 2D Fourier spectrum. The whole signal is analysed for its frequency content, describing the rhythmic variation of audible frequency throughout the audio data. It is assumed that although pitch is likely to vary within an audio signal, tempo will remain constant for any signal imported into the software tool

4.5.3 Synchronisation Option

Within each of these two analysis modes there is a synchronisation option that helps to define the conversion into two dimensions. Timbral mode offers tempo-synchronisation, which sets the frame size according to a note-length duration rather than the default duration in seconds. This may make it easier for individual notes within an audio signal to be separated.

In rhythmic mode, pitch-synchronisation is available, with the aim of improving the res-

olution of the audible frequency analysis. Each row/frame of the audio data is analysed to determine its pitch, and then the frame length is extended until the frame size is an integer number of periods of the fundamental pitch, hence removing effects of the rectangular window (section 4.3.3). This extended frame is then resampled to match the original frame size and analysis data for the frame is stored to ensure the process can be reversed. This process is based on the pitch-synchronous overlap-add technique [34].

4.5.4 Analysis Options GUI

When the user imports audio data into the software tool they are presented with a GUI that allows the analysis settings to be defined for that signal. This GUI changes appearance depending on the options chosen. The required parameters vary according to the mode and synchronisation settings, as demonstrated in figure 4.11.

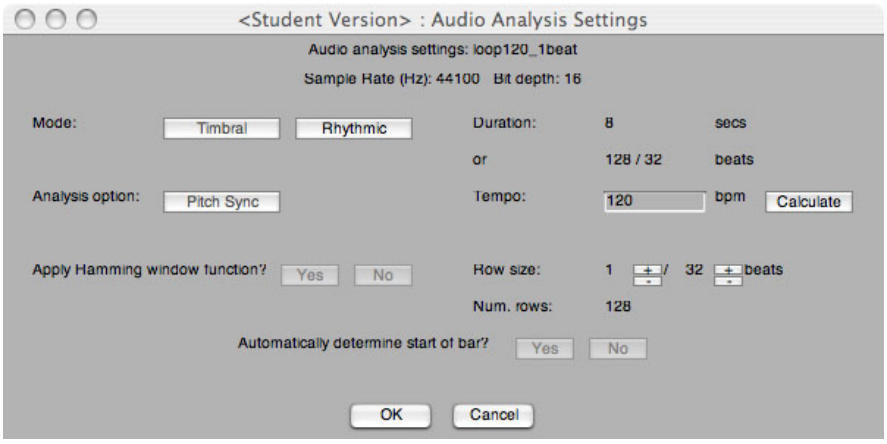
The function `analysis_settings` is called from `analyse_audio` to produce the analysis settings GUI and its M-file contains all of the object callbacks that define its functionality. All analysis options are stored in the `analysis_settings` structure within the `data` structure to be used subsequently to analyse the signal. The code and calculations in this function are all fairly trivial and have not been included here.

To allow mode selection, two push buttons were programmed to act as a button group where only one can be selected. The synchronisation option is presented as a push button programmed to toggle on and off. In timbral mode with the tempo-synchronisation off (figure 4.11b), the frame size can be entered in seconds using a text edit object. This was chosen because the frame size is continuous within the range of the signal duration. The text edit input is checked to ensure it is numeric using the `isnumber` function, which was written to perform this check for all text edit input within the software tool.

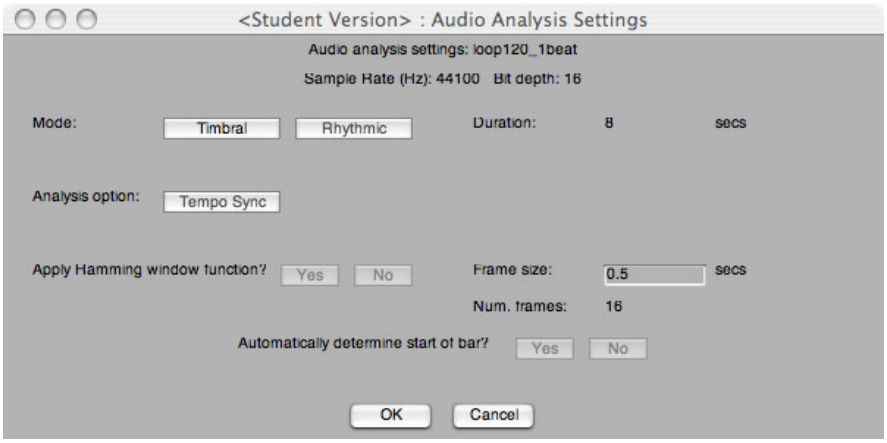
In either rhythmic mode (figure 4.11a) or tempo-synced timbral mode (figure 4.11c), the user can enter the tempo value of the signal using the text edit object or click the 'Calculate' button which uses the process described in section 4.7.2 to estimate the tempo of the signal. The row/frame size is defined in terms of a note duration chosen by the user. The software tool uses the American system of note names, where the note is defined in terms of fractions of a whole note or semibreve, since this is much easier to represent in software.

For the note duration setting, push buttons are provided to move the numerator and

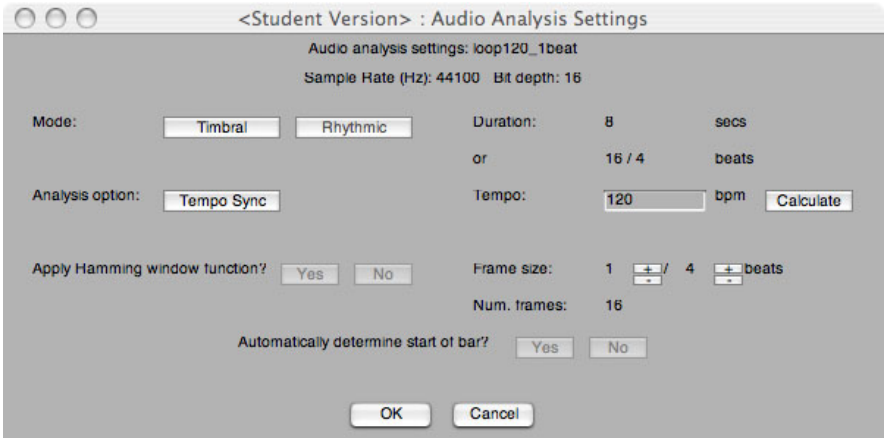
denominator values up and down. The denominator moves through an array of standard note divisions, from 1 (a semibreve) up to 128 (a semihemidemisemiquaver) which includes triplet divisions. The numerator simply allows a range from 1 up to the first integer that makes it longer than the signal duration.



(a) Rhythmic Mode



(b) Analysis Mode (Tempo Sync Off)



(c) Analysis Mode (Tempo Sync On)

Figure 4.11: Audio Analysis Options GUI

The GUI not only displays the analysis options but also supplementary information to help the user decide on appropriate settings. The signal duration is displayed in seconds and if in rhythmic or tempo-synced timbral mode it is also displayed as the exact number of the current note denomination. For example, at 120 bpm a signal duration of 8 seconds can also be displayed as 16/4 beats and at 110 bpm it would be 14.6667/4 beats. This can help the user determine the correct tempo, for example if they know the loop is exactly four bars long then the correct tempo setting would lead to a duration of 4/1 beats.

When the note division is changed, the numerator option is updated to get the nearest value to the previous frame size using the `update_ndivs` function. The function `update_duration` is then called to update the duration in terms of the current tempo division; this function is also called when the tempo value is changed. When in rhythmic or tempo-synced analysis mode, the user is only presented with tempo based frame/row size options, however the underlying `frame_size_sec` variable is used to store the frame size in terms of seconds whatever mode the analysis is in. If the tempo or the note numerator or denominator are changed then the actual frame size is recalculated using the `update_framesize` function. There is also a display of the number of frames that will be produced by the analysis options currently chosen, which is updated whenever the frame size is changed.

4.6 Analysis Implementation

Once the analysis options have been chosen, the `analyse_audio` function proceeds to obtain the 2D signal representations. The process of obtaining the raster image varies according to the analysis mode, and in rhythmic mode, the synchronisation option also determines between two processes. The implementation of each of the three options is described in detail in this section.

The first step in `analyse_audio`, before the raster image can be obtained, is to get the frame size parameter in terms of samples and store it in the `analysis` structure as `frame_size`. The following equation is used to extract the frame size in samples from the `frame_size_secs` variable:

$$\texttt{frame\_size} = \texttt{round}(\texttt{frame\_size\_secs} * F_s) \tag{4.21}$$

Where F_s is the sampling rate.

4.6.1 Timbral Mode

In timbral mode, the audio signal must first be divided into frames. This is done using the `rasterise` function, which can divide the audio array into a 2D matrix where each row contains a frame. Before each frame can be converted to a raster image, its pitch must be determined to define the image width. The `pitch_map` function, shown in listing 4.6 was written to take in the array of data frames and analyse the pitch of each one to produce the required variables.

```
    % calculate fundamental period values for each frame
    for frame = 1:handles.data.analysis.num_frames
        % display message box informing of pitch calc
        msg = ['Calculating pitch for frame ',num2str(frame),'/',num2str(
            handles.data.analysis.num_frames),'.'];
        hMsg = msgbox(msg,'Pitch Calculation','replace');
        % calculate pitch
        yin_vec = zeros(floor(handles.data.analysis.frame_size*0.5),1);
        period = yin(pitch_map_frames(frame,:),0.15,yin_vec);
        if period==0
            warning = {['The pitch for frame ',num2str(frame),' is
                undefined.'],...
                'A default period of 500 samples will be used.'};
            hWarnDlg = warndlg(warning,'Pitch detection error');
            uiwait(hWarnDlg);
            period = 500;
        end
        handles.data.analysis.period(frame) = period;
        handles.data.analysis.pitch(frame) = samples2freq(period,...
            handles.data.audio_settings.Fs);
        [handles.data.analysis.note(frame,:),handles.data.analysis.octave(
            frame)] =...
            freq2note(handles.data.analysis.pitch(frame));
```

```
        end
        delete(hMsg);
    end
```

Listing 4.6: Calculating Pitch For Frames Using `pitch_map`

The `yin` function, described in section 4.7.1, is used to determine the fundamental period of each frame and the results are stored in the `period` array which is in the `analysis` structure. The `yin` function can return a period of 0 when it fails to calculate the pitch, in which case the frame is given a default period of 500 and the user is warned. The pitch can then be set manually, after the initial analysis is complete. This is discussed in section 4.6.4.

The pitch information is made available to the user in terms frequency and musical note, so these are calculated in `pitch_map` too. Pitch frequency is calculated from the `period` data using the simple `samples2freq` function, which performs the following operation:

$$f = \frac{F_s}{n} \tag{4.22}$$

Where f is the pitch frequency in Hertz, n is the number of samples in the period and F_s is the sampling rate. This frequency value is stored in the `pitch` array within the `analysis` structure. Musical note value and octave number are determined from the pitch frequency using another utility function, `freq2note`. This function calculates the nearest MIDI note number [4] to the frequency, by this equation:

$$\text{midi\_note} = \texttt{round}(69 + 12 * \log_2(f/440)); \tag{4.23}$$

Matlab's `round` function rounds the input argument to the nearest integer. The note name and octave can easily be obtained from the MIDI note number and are returned to be stored in `note` and `octave` arrays within the `analysis` structure.

The `pitch_map` function also uses a `msgbox` object to inform the user that the pitch detection is in progress via a dialogue window, with a string that updates for each frame. This is because the pitch detection takes a few seconds to compute and the user needs to be informed of what is happening. The message dialogue is closed at the end of the function

when all calculations are complete. This technique is used several times in the software tool to inform the user of a current process.

Once the `pitch_map` function is complete the `period` array is used to define the image width settings for each frame, by rounding to the nearest integer. The `image` and `FT` cell arrays are initialised to the correct size according to the number of frames. Then the raster image representation is obtained for each frame, using the `calc_image` function. When in timbral mode, this function simply calls `rasterise` for each frame, with its associated image width parameter and stores the resulting image in the `image` cell array.

4.6.2 Rhythmic Mode Without Pitch-Synchronisation

This is the simplest process for obtaining the raster image, since only one image is required and there is no pitch detection. The `image` and `FT` variables are instantiated as 1-cell arrays and `imwidth` is set to equal the `frame_size` parameter. The `calc_image` function simply calls `rasterise` to convert the `audio` array directly into `image` with a width of `imwidth`.

4.6.3 Rhythmic Mode With Pitch-Synchronisation

When pitch-synchronisation is used in rhythmic mode, the conversion to a raster image ceases to be a one-to-one sample mapping. The process in `analyse_audio` is initially the same as for timbral mode. The signal is broken into frames, which are then analysed to determine their pitch. However the next stage is to calculate the data required for each frame and interpolate to fit it into the raster image. The `image` and `FT` variables are defined as 1-cell arrays, since in rhythmic mode there is only one raster image and one 2D spectrum. The raster image is first set to double the frame size to ensure the data is always upsampled preventing a loss of information and the image array is initialised with zeros. Then the `calc_image` function is called to fill the image array with data.

In `calc_image`, each row of the image data is calculated one by one. The required number of fundamental periods of data is obtained by finding the number of periods in the `frame_size` value and rounding up to the next integer, as in the following equation.

$$\texttt{num\_periods(frame)} = \texttt{ceil(frame\_size/period(frame))} \tag{4.24}$$

The frame data must then be located within the `audio` array and extracted to be interpolated to the width of the raster image, as shown in `calc_image` excerpt in listing 4.7. The Matlab function `interp1` is used to resample the frame data to fill a row of the image. It performs cubic spline interpolation on the data given the original sample points and the output sample points along the same signal. Cubic spline interpolation was chosen because it gives a lower interpolation error than polynomial interpolation for the same order of polynomial. The `frame` variable indicates the current image row being calculated. This code is contained within a `for` loop that increments `frame` for all row indices.

```
% set range indices for audio array
start = 1 + (frame-1)*handles.data.analysis.frame_size;
% length of frame has to be rounded up
len = ceil(handles.data.analysis.num_periods(frame)*handles.data
    .analysis.period(frame));
finish = start + len - 1;
Y = zeros(len,1);
% prevent going over the edge of the audio vector
if finish > length(handles.data.audio)
    finish = length(handles.data.audio);
    len = finish - start + 1;
end
% extract audio frame
Y(1:len) = handles.data.audio(start : finish);
% frame sample index values
x = linspace(1,length(Y),length(Y));
% work out sample indices for new resampled frame
xi = linspace(1, length(Y), handles.data.analysis.imwidth);
% interpolate the frame data to get resampled frame and add as a
% row in the image
Yi = interp1(x,Y,xi,'spline');
handles.data.image{1}(frame,1:length(Yi)) = Yi';
```

Listing 4.7: Calculating The Raster Image In Pitch-Synchronised Mode

4.6.4 Readjusting Analysis

Once the raster image data has been obtained, the `analyse_audio` function proceeds to determine the 2D Fourier transform data using `spec2D_init` as described in section 4.3.1. If there are 2D spectral transformations applied to the signal, they are then be processed; this is discussed in section 5.1. It then calls the `loaded` function to set up the GUI and display the data, at which point the importing of the audio signal is complete.

It is possible that the user might want to adjust the analysis options after observing the signal display. The main GUI presents the 'Reanalyse' button, which simply calls the `analyse_audio` function again to bring up the analysis settings GUI and allow the user to make adjustments to the settings. It proceeds through the new analysis process and redisplays the signal. The analysis settings GUI also contains a 'Cancel' button which aborts reanalysis and returns to the original settings, or if it is the initial analysis the software aborts the audio signal import.

After the signal analysis, each signal frame's detected pitch can be adjusted from within the 'Info' GUI window, shown in figure 4.12. This GUI displays signal information such as duration, data range, sampling rate and bit depth, as well as allowing observation and editing of frame specific pitch analysis information in timbral or pitch-synced rhythmic mode. The reason for this functionality is to correct any errors by the pitch detection algorithm and also allow experimentation with raster widths other than the fundamental period value.

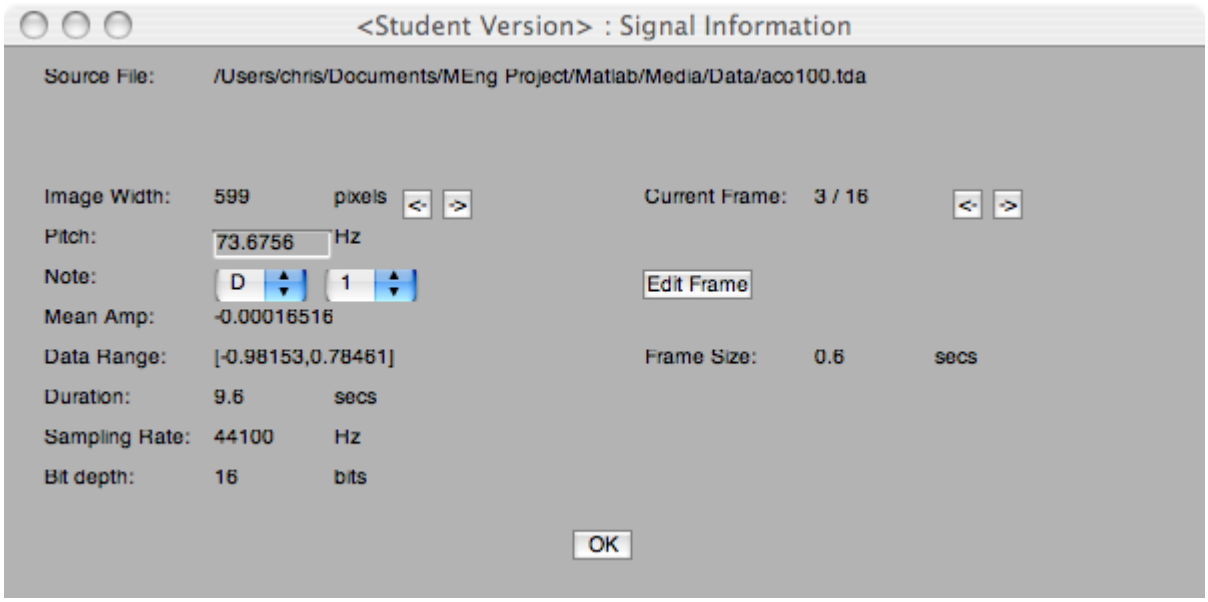


Figure 4.12: Info GUI Window Allowing Pitch Analysis Adjustment

The 'Edit Frame' button toggles between a static text display of the frame specific parameters and the editable objects shown in the figure. In timbral mode the analysis is adjusted in terms of image width, however in rhythmic mode there is no image width parameter, since this is fixed in the analysis settings GUI. Instead the underlying `period` parameter is used to adjust the signal analysis, which is calculated from the pitch frequency. Pitch frequency, note and image width are all linked, so when one is changed, the others must be recalculated. The utility functions `note2freq` and `freq2samples` were written to perform the inverse operation to the `freq2note` and `samples2freq` functions already introduced. When a new note value is selected by the user, the pitch frequency is calculated using `note2freq`, shown in listing 4.8.

```
function freq = note2freq(note,octave)
%NOTE2FREQ determines the frequency corresponding to a given note and
%octave.
% freq = note2freq(note,octave)

switch lower(note)
    case 'c ', num = 0;
    case 'c#', num = 1;
```

```
    case 'd ', num = 2;
    case 'd#', num = 3;
    case 'e ', num = 4;
    case 'f ', num = 5;
    case 'f#', num = 6;
    case 'g ', num = 7;
    case 'g#', num = 8;
    case 'a ', num = 9;
    case 'a#', num = 10;
    case 'b ', num = 11;
end

midi_note = num + 12*(octave+2);
freq = 440*2^((midi_note-69)/12);
```

Listing 4.8: Calculating Pitch Frequency From Musical Note In `note2freq`

The pitch frequency is calculated by using the MIDI note number to determine the ratio with A3 (note 69) at 440 Hz. The ratio between consecutive notes in the equal temperament scale is $2^{1/12}$. The period size is then determined using the `freq2samples` calculation:

$$n = F_s/f \tag{4.25}$$

Where the symbols have the same definition as in equation 4.22. In timbral mode, the image width is calculated by rounding the `period` value to the nearest integer, which is then stored in `imwidth`.

When the 'Info' GUI window is closed, if changes have been made to the analysis data, the `calc_image` function is run to recalculate the raster image representation and then `spec2D_init` is called to obtain the new 2D Fourier data. Finally `display_data` is called to redisplay the signal and store the settings as GUI data for the main window.

4.7 Feature Extraction

The audio signal analysis options incorporate the requirement for both pitch detection and tempo estimation algorithms to obtain accurate 2D signal representations. This section

describes the implementation of a robust fundamental pitch detection algorithm, the Yin algorithm [7], and the use of an autocorrelation algorithm to estimate tempo, provided by the MIRtoolbox [19].

4.7.1 Pitch Detection Using The Yin Algorithm

The YIN algorithm was implemented to detect the fundamental pitch period of audio data since it is known to produce accurate results. The algorithm is described in [7] and an implementation in the C language can be found at [3]. The M-file `yinpitch` was initially written to perform the YIN algorithm, however this process was much too slow to use in the software tool, taking several minutes for 15 seconds of audio data. Instead a MEX-file implementation was used, this is a subroutine produced from C source code. It behaves exactly like a built-in Matlab function being pre-compiled rather than interpreted at run time. Unlike a built-in Matlab function or an M-file, which are platform-independent, a MEX-file is platform specific and its extension reflects the platform on which it was compiled, in this case `.mexmaci`.

The function had to be written in C first and then compiled to the MEX-file in the Matlab environment. Microsoft Visual Studio was used to program and debug the file `yin.c` using the techniques described in [37]. In order to create a MEX file using C, a gateway function has to be written which determines how Matlab interfaces with the C code. It receives the number of input and output arguments specified when the function was called in Matlab, and an array of pointers to this data.These pointers are declared as type `mxArray` which is the C representation of a Matlab array. Several API routines are provided to allow access to the input data, these functions start with the prefix `mx*`. The variables required for the `yin` function are obtained using these API routines and then `yin` is called.

Listing 4.9 shows the gateway function of `yin.c`, which determines the pointers and values of the `mxArray` variables, so that they can be used in a call to the `yin` function. Note that the empty Matlab array `yin_vec` is passed in as an argument to use in the YIN algorithm calculations. The YIN algorithm itself is implemented as in [3] in the `yin` function which uses the given pointers to store its results in the `mxArray` variables.

```
/* the gateway function */
```

```
/* period_length = yin(input_vec, tolerance, yin_vec); */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  double *input_vec,*yin_vec, tolerance;
  double  *period_length;
  int yin_length;

  /*  check for proper number of arguments */
  if(nrhs!=3)
    mexErrMsgTxt("Three inputs required.");
  if(nlhs!=1)
    mexErrMsgTxt("One output required.");

  /* check to make sure the first input argument is a scalar */
  if( !mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
      mxGetN(prhs[1])*mxGetM(prhs[1])!=1 ) {
    mexErrMsgTxt("Input 'tolerance' must be a scalar.");
  }

  /*  get the scalar input tolerance */
  tolerance = mxGetScalar(prhs[1]);

  /*  create a pointer to the input matrices */
  input_vec = mxGetPr(prhs[0]);
  yin_vec = mxGetPr(prhs[2]);
  /*  get the length of the yin vector */
  yin_length = mxGetM(prhs[2]);

  /*  set the output pointer to the output matrix */
  plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
  /*  create a C pointer to a copy of the output value */
  period_length = mxGetPr(plhs[0]);

  /*  call the C subroutine */
  yin(period_length,input_vec,tolerance,yin_vec,yin_length);
}
```

Listing 4.9: The Gateway `mexFunction` In `yin.c`

The MEX-file was compiled using the command `mex yin.c` in the Matlab command line and the resulting `.mexmaci` file could then be used as if it were a normal Matlab function. The MEX implementation yielded approximately a 700% speed increase when compared to the M-file implementation.

4.7.2 Tempo Estimation With MIRToolbox

The MIRToolbox [20] was introduced in section 2.4, it provides a large variety of signal processing algorithms that extract information from audio signals. It was used in the 2D Fourier software tool to provide tempo estimation of audio signals with the `mirtempo` function. There are several tempo detection methods offered in the MIRtoolbox but the autocorrelation option was the most reliable in initial experiments, where other methods either caused errors or were quite inaccurate. The algorithm operates by obtaining an onset detection curve of the audio data and then computing the autocorrelation function of this data.

```
% calculate tempo
miranswer = mirtempo([handles.source.audio_path,handles.source.file '.
    wav'],'Autocor');
answer = mirgetdata(miranswer);
```

Listing 4.10: Determining Tempo Using `miraudio`

Listing 4.10 shows the use of `mirtempo` from within the 'Calculate' button callback in `analysis_settings`. The MIRtoolbox requires that the audio data is within a file on disk, hence path of the source WAVE-file is passed in as an argument. The functions of MIRtoolbox return their results encapsulated within their own class-type, so the data is extracted using the provided `mirgetdata` function.

4.8 Visual Analysis Tools

Several tools have been provided within the software to enable more thorough plot analysis. They are described in this section.

4.8.1 Plot Zoom and Pan

The zoom and pan tools are provided in the default figure menu and toolbar layout to allow the user to examine aspects of a plot display more closely. Matlab functions were written to reproduce this default zoom and pan functionality, allowing these tools to be incorporated into the 2D Fourier software whilst using a custom menu and toolbar layout. This was vitally important since each of the axes on the main GUI represent a large amount of data that could not be fully observed at normal resolution. Matlab provides `zoom` and `pan` 'mode' objects that can be applied to a particular figure and allow the behaviour of these tools to be customised using settable properties.

The tools can be horizontal, vertical or unconstrained as in a default Matlab figure and the zoom tool can go in or out. These options are available to the user through the 'Plot Tools' menu, and the callback functions within `app.m` set the properties of the `zoom_obj` and `pan_obj` appropriately. The tools can be toggled on and off from both the toolbar and the 'Plot Tools' menu.

4.8.2 Data Cursor

The data cursor tool is also provided in the default Matlab figure, however this tool has been extended to provide a custom data tip according to the particular plot, as shown in figure 4.13. This allows the user to extract the precise data value represented by a plot point.

The `datacursormode` object is instantiated in the same way as `pan` and `zoom`, by passing in a figure handle. It's properties can then be set to define the behaviour of the data cursor tool on that particular figure. This tool can again be toggled on and off from either the toolbar or the 'Plot Tools' menu, and the option to view the data cursor as a pointer or in a window is provided, as in the default Matlab figure. The '`UpdateFcn`' property is set in the same way as a callback (section 3.4.2) to indicate the function that defines the custom data cursor.

The `dcm_updateFcn` function is called when the data cursor tool is used on any plot within the main GUI window. It is provided with the `event_obj` argument which identifies the point indicated by the cursor. The handle of the plot axes can be obtained by getting the

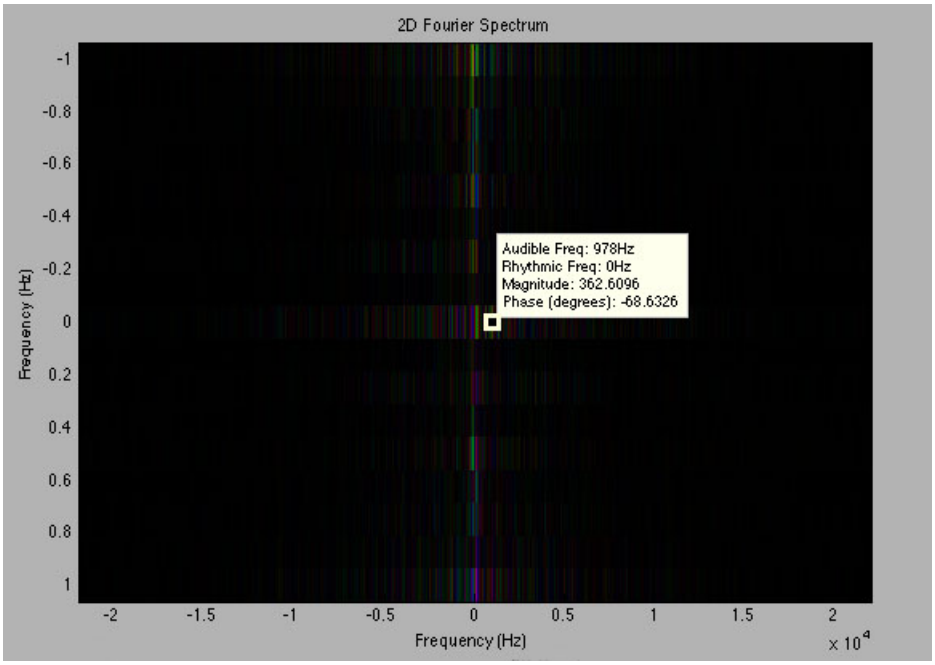


Figure 4.13: Custom Data Cursor

`event_obj`'s '`Target`' item and inspecting this item's '`Parent`' property. The '`Position`' property of `event_obj` can be used to identify the indices of the cursor pointer within the data, this is stored in the local `pos` variable.

For the audio waveform plot, the time and amplitude values can be displayed directly from the `pos` array. The 1D Fourier spectrum is almost as simple, the magnitude and frequency data can be displayed directly from `pos` but it was decided that phase should be displayed in degrees rather than the π*radians scale used on the display, because many people are more familiar with the degree representation.

The raster image plot provides four values in the data cursor, the (x,y) co-ordinates of the cursor within the image, and also the time and amplitude settings, demonstrating the relationship with the audio waveform. Time is calculated using the sample rate and image width parameters, in the following equation:

$$\text{time\_index} = ((\text{pos}(2) - 1) * \text{imwidth}(\text{current\_frame}) + \text{pos}(1))/\text{Fs} \tag{4.26}$$

Where Fs is the data sample rate. The data structure prefixes have been removed from

the variable names for clarity. The amplitude is obtained from the raster image data in the `image` cell array, using the `pos` co-ordinates.

The data cursor for the 2D Fourier spectrum is slightly more complex. The `pos` vector contains the frequency values of the cursor point, so the indices of the data matrix must be obtained using Matlab's `find` function. The magnitude and phase components are displayed in the data cursor, which have to be obtained from the RGB display data using the `colour2polar` function which performs the inverse operation to `polar2colour`. The brightness and contrast functions alter the display, therefore they are incorporated into the calculations. An excerpt from `dcm_updateFcn` is given in listing 4.11, it shows the calculation of the data cursor display for the 2D spectrum.

```
xdata = get(target,'XData');
ydata = get(target,'YData');
x = find(xdata>pos(1),1,'first')-1;
y = find(ydata>pos(2),1,'first')-1;
% when only one row in spectrum
if isempty(y)
    y = 1;
end
data = get(target,'CData');
value = data(y,x,:);
[magLN,phase]=colour2polar(value,...
    handles.plot_settings.specBrightness,...
    handles.plot_settings.spec2D_mode);
magN = (2.^nthroot(magLN,handles.plot_settings.specContrast) - 1);
mag = magN*handles.data.spec2D_settings.max_mag(...
    handles.frame_settings.current_frame);
txt = {['Audible Freq: ', num2str(pos(1)),'Hz'],...
    ['Rhythmic Freq: ', num2str(pos(2)),'Hz'],...
    ['Magnitude: ', num2str(mag)],...
    ['Phase (degrees): ', num2str(phase*180/pi)]};
```

Listing 4.11: Custom Data Cursor For 2D Fourier Spectrum

The output of the `dcm_updateFcn` is the `txt` variable, which is displayed in the data cursor window. The output is varied according to the display mode for the 2D spectrum, however, for demonstration purposes, only the output for the combined magnitude and phase display

is shown in this listing.

4.8.3 2D Fourier Spectrum Legend

A custom legend tool has been developed for the 2D Fourier spectrum display, which appears in a stand-alone GUI window. It was decided that although the existing tools ensured a flexible display, the normalisation and scaling of data abstracted too much from the actual values. This legend (in figure 4.14) shows a complex unit circle representing the logarithmic, normalised scaling of data and it has it's own data cursor object that displays the magnitude and phase value represented by a particular colour. It aims to put the magnitude values of the 2D spectrum in context of the data range and demonstrate the phase angle in terms of the colour wheel.

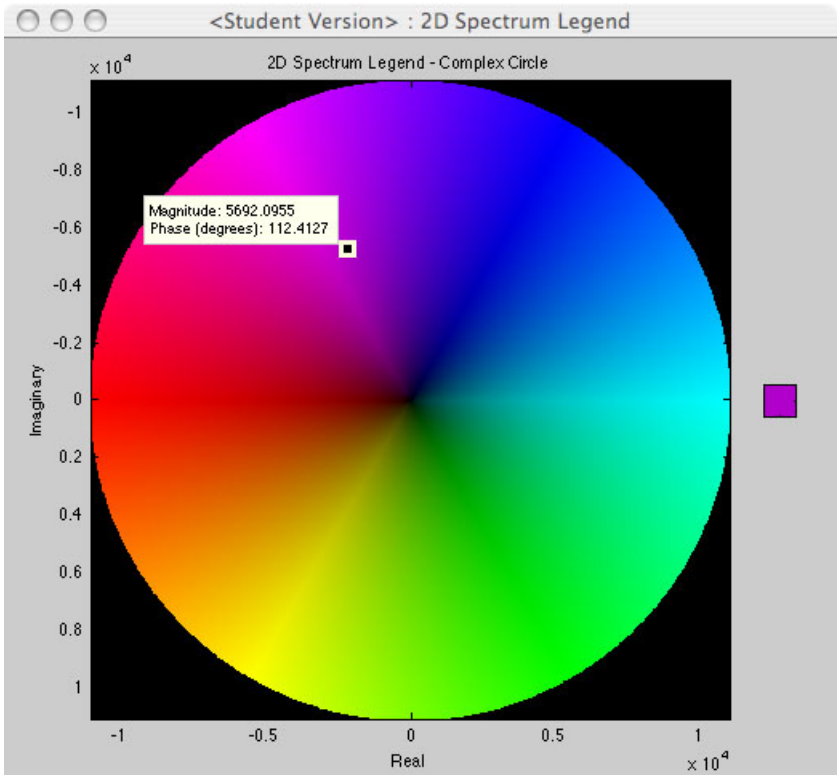


Figure 4.14: 2D Fourier Spectrum Legend Window

The legend figure uses a seperate `datacursormode` object, however the same '`UpdateFcn`' is used. The legend data cursor shares the code for the 2D spectrum display in listing

4.11 however it's pointer doesn't have any frequency values. It also displays the selected colour in the small plot to the right hand side of the unit circle. The legend plot is updated accordingly when brightness and contrast parameters are adjusted.

The complex unit circle data is stored in a 2D matrix of size 500∗500, with any points outside the circle having 0 value. This matrix is stored on disk in a MEX-file and loaded during the `app_OpeningFcn` before the main GUI appears. When the legend is plotted using the `plot_legend` function, the complex matrix is converted to a RGB colour representation by the same process as the `FT` data in `calc_spec2D`, although starting from the `magN` variable.

4.9 Resynthesis

Once a comprehensive analysis tool had been developed, the next stage of the investigation was to perform signal transformations by manipulating the 2D Fourier data. This meant it had to be possible to obtain the other signal representations from the 2D Fourier domain representation. The `resynth` function was written to resynthesise the time domain data.

First the maximum magnitude array (`max_mag`) must be updated for the new Fourier data, using the `calc_spec2D` function. The raster image representation can easily be obtained using Matlab's 2D inverse FFT function `ifft2` upon the `FT` data. The result of `ifft2` is likely to produce a complex result but the imaginary component is due numerical rounding errors and can simply be removed using Matlab's `real` function [16]. The `width_pad` and `height_pad` variables in the `spec2D_settings` structure must be inspected for each frame and the padded row and/or column should be removed if present to achieve the correct raster image representation.

The method by which the data is converted from raster image to audio depends upon the analysis process followed, as described in section 4.6. The parameters in the `analysis_settings` structure are inspected to determine the correct synthesis process.

4.9.1 Derasterisation

All resynthesis methods in the software tool use the `derasterise` function to obtain a 1D array from a 2D representation. The function incorporates the `hop` parameter to allow image rows to overlap within the 1D array representation. This option is required in pitch-synced rhythmic mode, however to perform the standard derasterisation process, as described in [39], the `hop` parameter should be set to the width of the `image` array. This maintains a one-to-one sample mapping between input and output. The code for `derasterise` is given in listing 4.12.

```
function array = derasterise(image,hop)
%DERASTERISE converts between data from a 2D representation to a 1D
%representation using raster scanning.
%
%This function can be used to convert from a grayscale image to a
%monophonic audio signal.
%
%image = the 2D data representation
%
%array = the 1D data representation

    %calculate the required output array size
    imsize = size(image);
    arraysize = imsize(2) + (imsize(1)-1)*hop;
    %create an empty array
    array = zeros(arraysize,1);
    %derasterise data
    for i = 1:imsize(1)
        arrayindex = (i-1)*hop;
        array(arrayindex+1:arrayindex+imsize(2)) = ...
            array(arrayindex+1:arrayindex+imsize(2)) + ...
            image(i,1:imsize(2))';
    end
```

Listing 4.12: Derasterisation Process

4.9.2 Timbral Mode

When the signal analysis is in timbral mode, the conversion from raster image representation to the audio array is simple. Each frame is converted to a 1D array by derasterising its image with the correct image width value and these arrays are stored in a 2D array, equivalent to the `pitch_map_frames` matrix in `analyse_audio`. The complete audio array is calculated by derasterising this 2D array using the `frame_size` parameter to define the hop, which is equal to the width of the 2D array

4.9.3 Rhythmic Mode Without Pitch-Synchronisation

For rhythmic mode without pitch-synchronisation there is only one raster image and it is simply derasterised with no row overlap to obtain the audio array, i.e. `hop` is set to the image width (`imwidth`).

4.9.4 Rhythmic Mode With Pitch-Synchronisation

When pitch-synchronisation is used, each row of the raster image has to be resampled to the original audio sampling rate before the audio array is created. Listing 4.13 shows an excerpt from `resynth` that resamples each row of the raster image by cubic spline interpolation and stores it in an intermediate array `tempo_frames`. The interpolation is performed using two arrays of sample indices, as in `calc_image`, the forward process. Array `x` corresponds to the sample points of the source data, in this case the upsampled data, and `xi` gives the required sample points for the interpolated output data, which is at the audio sample rate in this case. To create arrays `x` and `xi`, the size of the frame at the original audio sample rate must first be calculated using the period size and the number of periods contained in the frame. The frame can then be resampled and stored in the `tempo_frames` array.

```
        % rhythmic mode, pitch-synced
        % resample each row by interpolation
        height = size(handles.data.image{1},1);
        width = size(handles.data.image{1},2);
        tempo_frames = zeros(height,handles.data.analysis.frame_size*2);
```

```
        for frame = 1:height
            Y = handles.data.image{1}(frame,:);
            num_samples = ceil(handles.data.analysis.period(frame)*...
                handles.data.analysis.num_periods(frame));
            x = linspace(1,num_samples,width);
            xi = linspace(1,num_samples,num_samples);
            tempo_frames(frame,1:num_samples)=interp1(x,...
                handles.data.image{1}(frame,:),xi,'spline');
        end
        handles.data.audio = derasterise(tempo_frames,...
            handles.data.analysis.frame_size);
```

Listing 4.13: Calculating `audio` From The Raster Image In Pitch-Synchronised Mode

Once all of the frames have been resampled, the `tempo_frames` array is derasterised to produce the audio signal. This is when the `hop` argument of `derasterise` is utilised. By setting it to `frame_size`, the rows of the `tempo_frames` array can be inserted at the correct position in the audio array, even though the frames of data are all of different length. The frames will overlap, since they are all longer than `frame_size`, which will cause signal discontinuities. This issue could be addressed by the use of windowing and setting the amount of overlap to the window functions half-power point, as with the STFT (section 2.2.2). However, it was decided that for the purposes of analysis and signal transformation in the software tool, the pitch-synchronisation technique did not provided much additional benefit. The resampling of each row individually means that the 2D spectrum display axes do not give the correct frequency scale for any point. The design of signal transformations would also have to be much more complex to properly compensate for the resampling of row data. As a result the pitch-synchronous analysis and synthesis were not developed any further.

Chapter 5

Two-Dimensional Fourier Processing of Audio

The second major phase of the project was to investigate methods of audio signal transformation by processing 2D Fourier data using the knowledge gained from the prior analysis investigation. This processing has two uses, to provide novel musical transformations for creative sound design and to further the understanding of 2D Fourier analysis. The processes used to obtain signal transformations should be viewed as prototypes. They were designed with the aim of investigating the potential of techniques, based on the initial understanding developed to that point. This is the first part of an iterative process of analysis and refinement, that would continue to further the understanding of 2D Fourier processing and lead to the development of more useful and robust signal transformations.

A set of signal transformations have been developed and integrated into the 2D Fourier software tool. This chapter will describe the extension of the software to allow 2D Fourier processing of audio, and discuss each the implementation of each transformation individually. Chapter 6 will discuss the results of audio signal processing using these transforms and analyse their operation.

5.1 2D Fourier Processing Overview in MATLAB Tool

The preliminary investigation of 2D Fourier domain signal processing was performed using Matlab's debugging tools. The Fourier data was altered in the command line and then the signal was resynthesised using the functions developed in section 4.9. However it was soon apparent that the architecture of the software had to be extended to allow efficient investigation. Section 3.1.3 details the software requirements that were determined to allow signal transformation.

Users can apply a chain of various transformations to the signal data, using a GUI that mimics the style of DAW plug-ins. It was decided to limit the maximum number of transformation processes to 5 to prevent excessive use of memory and long processing times. It is assumed that a user would rarely require more than 5 transformation processes at once.

Many of the available processes have adjustable parameters that can be set by the user when the process is applied and also readjusted at any time. This processing is not in real-time so the transformations are destructive, losing the original signal information. A copy of the original unprocessed data has to be stored to allow this process parameter adjustment, giving the user the impression that processing is non-destructive. The processed data is recalculated from this original data each time a new transformation process is added, removed or adjusted. The user can also switch the display between the original and unprocessed signals to analyse the effects of the signal transformations.

5.1.1 Data Structures

After an audio signal is imported into the software tool and analysed, as described in section 4.6, the `data` structure, containing the signal representations and analysis data, is copied into the `original_data` structure. This is done at the end of `analyse_audio` just before the `loaded` function is called. The `original_data` structure maintains a copy of the unprocessed data, and is only overwritten when the analysis settings are changed by calling `analyse_audio` again (section 4.6.4).

The `data` structure stores the current data representation, as displayed in the main GUI. If the user chooses to view the original unprocessed data then `original_data` is copied

directly into `data`. If the processed data is to be displayed then `data` is obtained by running `original_data` through the chain of 2D Fourier transformation processes.

The details of the processes are stored in the `processes` structure, which is shown in figure 5.1. It contains the integer variable `num_proc`, which indicates how many processes are currently in the chain, and a 1D array of *struct* objects of the size of `num_proc`. This array, called `process`, stores the details of each process in the chain in the order they should be implemented. Each structure in `process` has the `type` variable which contains the name of the process it represents, and the `bypass` variable which allows a process to be removed from the processing chain temporarily, without losing its settings. The rest of the variables used in each structure are particular to the specific transformation process.

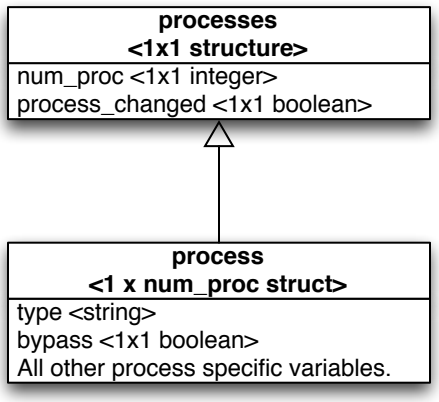


Figure 5.1: 2D Spectral Processes GUI Window

The names of all of the processes in the software tool are stored in the cell array `proc_names` within `handles` because they are used at several points within the software and it was much easier during implementation to add or change process names when they were all defined in one place. Each process name is accessed by calling `proc_names` with the array index of that process' name.

5.1.2 2D Spectral Processes Window Design

Once a signal has been loaded and displayed in the main GUI window, the '2D Spectral Processes' window is created using the function `create_proc_panel`. This window, shown

in figure 5.2, allows the user to add and remove signal transformation processes to the processing chain in a similar style to DAW plug-ins. It is displayed by clicking the 'Processing' button in the main GUI.

When the '2D Spectral Processes' window is created,`create_proc_panel` first determines the number of buttons required in the window. This corresponds to the number of transformation processes in the chain, plus an extra 'Empty' button underneath to allow a new process to be added. If the audio signal has just been imported into the software then there will only be an 'Empty' button, as in figure 5.2a, however a signal can also be loaded as a TDA-file with the process chain data, such as in figure 5.2b. The GUI figure's height and position change according to the number of buttons displayed, so they are calculated in `create_proc_panel` before the *figure* object and the required buttons are created.

(a) No Transformation Processes Applied (b) Chain of Four Different Processes

Figure 5.2: 2D Spectral Processes GUI Window

The process buttons are labelled with their process name, as given in `proc_names`, and displayed in a column in the order that the processes are stored in the `process` array. The 'Empty' button is placed at the bottom of the column and is disabled. Each process button has a small 'options' button next to it, which opens a context menu when clicked. The context menu reveals the list of available transformation processes, which are divided into two groups, adjustable and fixed, according to whether the process has adjustable parameters or not.

The object handles for these GUI components are stored in the `proc_popup` structure, which is added to `handles` and stored as GUI data in the '2D Spectral Processes' figure object. Throughout the software tool, when GUI data is updated, it is stored in both the main GUI and the processes GUI to ensure that the transformation process functions can

access the signal data. The processes figure is closed when the data is reanalysed (hence a new figure is created) or the software tool is closed. When the user chooses to close the figure window, it is only hidden temporarily.

5.1.3 2D Spectral Processes Window Functionality

The function `proc_popup_callbacks` sets and contains the callback functions for objects within the '2D Spectral Processes' window, as shown in listing 5.1; it is called at the end of `create_proc_panel`. It can be seen that the context menus, process buttons and option buttons are all stored in arrays with the length of `num_buttons`. Therefore it is clear which item in the `process` array each object corresponds to.

```
for num = 1:handles.proc_popup.num_buttons
    for ptype = 1:size(handles.proc_popup.cmenus(num).proc,2)
        set(handles.proc_popup.cmenus(num).proc(ptype),'Callback',...
            {@cmenu_clicked,num,ptype},'Interruptible','off');
    end
    set(handles.proc_popup.buttons(num),'Callback',{@button_clicked,...
        num},'Interruptible','off');
    set(handles.proc_popup.opt_buttons(num),'Callback',{...
        @opt_button_clicked,num},'Interruptible','off');
end
set(handles.proc_popup.figure,'CloseRequestFcn',{@proc_popup_close});
```

Listing 5.1: Setting Object Callbacks In The 2D Spectral Processes Window

The same callback function is used for all the process button objects in the `buttons` array, the index of the clicked object is simply passed as the argument `num` into the function. The same technique is used for both `opt_buttons` which contains the option buttons objects.

The `cmenus` array contains a structure for each process in the `process` array, containing the handles for all objects in the menu. Each *uimenu* object corresponding to a transformation process has its handle stored in the `proc` array, within each `cmenus` structure. The index of these handles corresponds to the index of each process name in the `proc_names`. The function `create_context_menu` is called from `create_proc_popup`, for every menu required. The `num` argument is passed in, so the menu's data can be stored in the correct entry in

the `cmenus` array.

The callbacks for each item in the context menus are defined in `proc_popup_callbacks` also. Listing 5.1 shows how the callback function `cmenu_clicked` is used for every menu item on every menu, and this callback receives the arguments `num` and `ptype` to distinguish which menu was used and what process was selected.

When the user clicks an option button, the `opt_button_clicked` callback displays the corresponding context menu, as shown in listing 5.2. The layout of each context menu is identical. The top-level offers the options 'Empty', 'Adjustable' or 'Fixed'. 'Adjustable' and 'Fixed' bring up sub-menus offering a list of transformation processes, which have either adjustable or fixed parameters.

```
function opt_button_clicked(hObject,eventdata,button_num)
    handles = guidata(gcf);
    button_pos = get(handles.proc_popup.opt_buttons(button_num),'Position');
    set(handles.proc_popup.cmenus(button_num).h,'Position',...
        [button_pos(1) button_pos(2)],'Visible','on');
end
```

Listing 5.2: Displaying The Context Menu

The `cmenu_clicked` function is called when any item on a context menu is clicked. Its actions depend on the option clicked and the item in the `process` array that the menu relates to. If an item in the bottom context menu, which corresponds to the 'Empty' button, is clicked, the process chosen will simply be added at the end of the chain in the `process` array. If this process has adjustable parameters, then it will display a window requesting the values of these parameters before it is added. Once added, the relevant button's string must be changed to match the process name, the GUI window resized and a new item must be created in the `button`, `opt_button` and `cmenus` arrays. The buttons are then displayed at the bottom of the window, with the process button deactivated and displaying the string 'Empty'. The function `add_button` creates and displays the new button and menu objects, after resizing the figure and moving the other GUI objects to the correct position. It is shown in listing 5.3 as a demonstration of the adjusting of the GUI layout that is common in the design of this window.

```
function handles = add_button(handles)
    if handles.proc_popup.num_buttons < 5
        cur_pos = get(handles.proc_popup.figure,'Position');
        set(handles.proc_popup.figure,'Position',[cur_pos(1) ...
            cur_pos(2)-10 cur_pos(3) cur_pos(4)+20]);
        cur_pos = get(handles.proc_popup.title,'Position');
        set(handles.proc_popup.title,'Position',[cur_pos(1) ...
            cur_pos(2)+20 cur_pos(3) cur_pos(4)]);
        for num = 1:handles.proc_popup.num_buttons
            cur_pos = get(handles.proc_popup.buttons(num),'Position');
            set(handles.proc_popup.buttons(num),'Position',[...
                cur_pos(1) cur_pos(2)+20 cur_pos(3) cur_pos(4)]);
            cur_pos = get(handles.proc_popup.opt_buttons(num),'Position');
            set(handles.proc_popup.opt_buttons(num),'Position',...
                [cur_pos(1) cur_pos(2)+20 cur_pos(3) cur_pos(4)]);
        end
        handles.proc_popup.num_buttons = handles.proc_popup.num_buttons + 1;
        handles = create_context_menu(handles,...
            handles.proc_popup.num_buttons);
        handles.proc_popup.buttons(handles.proc_popup.num_buttons) = ...
            uicontrol(handles.proc_popup.figure,...
              'Style','pushbutton',...
              'UIContextMenu',handles.proc_popup.cmenus(...
                  handles.proc_popup.num_buttons).h,...
              'Enable','off',...
              'String','Empty',...
              'Position',[100,20,90,15]...
            );
        handles.proc_popup.opt_buttons(handles.proc_popup.num_buttons) = ...
          uicontrol(handles.proc_popup.figure,...
              'Style','pushbutton',...
              'String','\/',...
              'Position',[190,20,10,15]...
          );
        proc_popup_callbacks(handles);
        guidata(gcf,handles);
    end
end
```

Listing 5.3: Adding a Button To The Processes GUI

When the context menu corresponds to an existing process then there are three options. If 'Empty' is chosen, the process is removed from the `process` array and the corresponding process and option buttons are deleted. The GUI window is resized and any buttons below those deleted are moved up. Listing 5.4 shows how a process is removed from the `process` array using the `remove_process` function.

```
function handles = remove_process(handles,num)
    handles.processes.process(num:handles.processes.num_proc-1) = ...
        handles.processes.process(num+1:handles.processes.num_proc);
    handles.processes.num_proc = handles.processes.num_proc - 1;
    handles.processes.process = handles.processes.process(1:...
        handles.processes.num_proc);
    if handles.processes.num_proc == 0
        set(handles.tb_viewProc,'Enable','off');
        set(handles.tb_viewProc,'State','off');
        set(handles.tb_viewOrig,'State','on');
    end
end
```

Listing 5.4: Removing a Transformation Process Using `remove_process`

If the current process, which is checked/ticked on the menu, is selected, that process' settings window will be displayed for an adjustable process allowing the parameters to be altered; for a fixed process no action will occur. This functionality is identical to the `button_clicked` callback, which is called when a process button from the `buttons` array is clicked.

If a new process is selected, this will replace the currently existing process. This is the same as creating a new process, but the array index is set so that the old data is overwritten in the various arrays and no new button is added to the GUI.

5.1.4 Components of a Generic Transformation Process

Section 5.1.2 introduced the fact that some transformations have been designed to perform a fixed process upon the 2D Fourier data, whilst others have adjustable parameters. All transformation processes have a 'run' function that has the prefix `process_*` followed by the process name (a short version that doesn't necessarily match the name in `proc_names`). This function performs the processing on the `FT` data.

Adjustable processes also have 'create' and 'adjust' functions with the prefixes `create_*` and `adjust_*`. The 'create' function initialises the required parameter variables in the process' structure within the `process` array and these variables are given initial default values. It is called when the user first creates the process by choosing it from a context menu within the processing window. Listing 5.5 shows the 'create' function for a generic process. The `type` variable is set to the appropriate string in the `proc_names` array.

```
function handles = create_process(handles,proc_num)
    handles.processes.process(proc_num).type = ...
        handles.proc_names{correct_index};
    handles.processes.process(proc_num).bypass = false;
    handles.processes.process(proc_num).process_specific_var1 = default1;
    handles.processes.process(proc_num).process_specific_var2 = default2;

    handles = adjust_process(handles,proc_num);
end
```

Listing 5.5: Creating a Generic Process

At the end of every 'create' function, the process' 'adjust' function is called to allow the user to set the initial settings as they desire. The 'adjust' function contains the code to create and display a GUI window that presents the current parameter settings of the GUI, as stored in `process(proc_num)`, and allows the user to adjust them. An 'adjust' function for a generic process is shown in listing 5.6.

```
function handles = adjust_process(handles,proc_num)
    handles.cur_process = handles.processes.process(process_num);
    handles.cur_process_num = process_num;
    handles.processes.process_changed = false;
```

```
    %create figure and components, storing the object handles in }handles.
        proc_name_popup.
    %setting display to present current parameter settings...
    %...then
    handles.cur_process.bypass = ~handles.cur_process.bypass;
    guidata(handles.proc_name_popup.figure,handles);
    bypass_Callback(handles.figure1,'');
    handles.cur_process.bypass = ~handles.cur_process.bypass;

    guidata(handles.proc_name_popup.figure,handles);
    proc_name_callbacks(handles);
    set(handles.proc_name_popup.figure,'Visible','on');
    uiwait(handles.proc_name_popup.figure);
    handles = guidata(handles.proc_name_popup.figure);
    delete(handles.proc_name_popup.figure);
    handles = rmfield(handles,'cur_process_num');
    handles = rmfield(handles,'proc_name_popup');
    handles = rmfield(handles,'cur_process');
end

function proc_name_callbacks(handles)
    set(handles.proc_name_popup.figure,'CloseRequestFcn',{@my_closefcn},...
        'Interruptible','off');
    set(handles.proc_name_popup.bypass_button,'Callback',{@bypass_Callback
        },...
        'Interruptible','off');
    % any other object callbacks required...
end
```

Listing 5.6: Adjusting Parameters of a Generic Process

The current process' structure and its index in the `process` array are stored within the `handles` structure so that they can be easily accessed in callback functions. The Boolean `process_changed` variable is used to indicate whether any parameters are changed by the user, so that the software tool can determine whether the processed data needs to be recalculated (section 5.1.5).

The design originally incorporated a cancel option into each process' parameter adjustment

figure, so the user could decide to abandon the changes and retain their old settings. This option was deemed unnecessary and removed from the software tool in the later stages of development. Due to this original design, a copy of the current process' data structure is stored within the `handles` structure as `cur_process` to be edited locally and the process' index within the `process` array is also stored as `cur_process_num`, allowing the process data to be reinserted into `process` when adjusting is complete.

The bulk of the `adjust_*` function code will create the GUI figure and its objects, which are specific to the particular process. However, every process contains a 'Bypass' button which allows the user to temporarily bypass the processing of that particular transformation when the processes are run. This buttons callback function enables or disables the editable components of the figure and inverts the value of the `bypass` variable. This callback is utilised in the `adjust_*` function to set enable/disable the GUI components appropriately according to the initial state of `bypass`, which requires the variable to be inverted before `bypass_Callback` is called.

The `adjust_*` function then stores the `handles` structure as its figure's GUI data, and calls a local function to set the callbacks of the GUI objects, as well as the figure's `CloseRequestFcn` which determines the actions performed on closing the figure window. Finally it makes the figure visible to the user and then calls Matlab's `uiwait` function with the 'adjust' figure's handle as an argument. This function blocks program execution in the code sequence until the `uiresume` function is called with the same figure handle.

Use of `uiwait` and `uiresume` means that the parameters of the process can be adjusted by the user, whilst the `adjust_*` function is frozen. When the user closes the figure window, the function defined in its `CloseRequestFcn` property is called, which is set to `my_closefcn` shown in listing 5.7.

```
function my_closefcn(hObject,eventdata)
    handles = guidata(gcf);
    handles.processes.process(handles.cur_process_num)=handles.cur_process;
    guidata(handles.proc_name_popup.figure,handles);
    uiresume(handles.proc_name_popup.figure);
end
```

Listing 5.7: The 'Adjust' Figure Window's `CloseRequestFcn`

This function stores the `cur_process` data in the `process` array at the index `cur_process_num`, overwriting the previous settings for the transformation process. The `handles` structure is then stored as GUI data with the 'adjust' figure and the `uiresume` function is called, allowing the execution of the `adjust_*` function to continue.

Once the user has closed the figure window, the `adjust_*` function obtains the `handles` structure using `guidata`, then deletes the figure and data concerned with the current process, as shown in listing 5.6. The `adjust_*` function then returns the `handles` variable with the process parameters stored in the data structure `process(proc_num)` and the Boolean variables `process_changed` which states whether any of the variables have been altered.

5.1.5 Running The Processes

As introduced in section 5.1.3, the parameters of a process can be adjusted via either the `cmenu_clicked` or `button_clicked` callback. The relevant `adjust_*` function is called to display a figure allowing the user to observe and set the process parameters, and further program execution is halted until this figure has been closed (section 5.1.4). Only then will the `adjust_*` function return the `handles` data structure to the calling function. Both `cmenu_clicked` and `button_clicked` inspect the value of the `process_changed` variable to determine whether the process parameters have been adjusted as shown in listing 5.8.

```
if handles.processes.process_changed
    handles = rmfield(handles,'process_changed');
    processes_changed(handles);
end
```

Listing 5.8: Determining If Process Parameters Have Been Adjusted

If a process has been created or adjusted the `processes_changed` function is called, which calls `run_processes` to obtain the new processed signal data and then `display_data` to display the new signal representation on the main GUI.

The `run_processes` function copies the unprocessed data from `original_data` to `data`, then loops through each process in the `process` array calling the correct `process_ function` to run the transformation process, altering the 2D Fourier domain data in `FT`. The process

type is identified using a `switch...case...` statement to compare the value of the `type` variable with each string in the `proc_names` array.

Once all processes have been run, the `resynth` function is called to generate the raster image and audio data representations from the 2D Fourier domain data, as described in section 4.9.

5.1.6 Observing Original and Processed Data

The main GUI's toolbar (section 3.5) contains the 'View Original Signal' and 'View Processed Signal' buttons, allowing the user to switch the display between the unprocessed and processed signal representations. If no processes are defined, the 'View Original Signal' button on the main GUI's toolbar is selected and the 'View Processed Signal' button is deactivated. When a process is added, the processed result is displayed and the 'View Processed Signal' button is activated and selected.

Section 5.1.1 described the use of the `original_data` and `data` structures to store the unprocessed data and the currently displayed data. When the 'View Original Signal' button is selected, it's callback function `tb_viewOrigCallback` is called. It is shown in listing 5.9.

```
function tb_viewOrigCallback(hObject, eventdata)
    handles = guidata(gcf);
    if strcmp(get(handles.tb_viewProc,'State'),'off')
        set(handles.tb_viewOrig,'State','on');
    else
        set(handles.tb_viewProc,'State','off');
    end
    if handles.processes.num_proc > 0
        set(handles.process_button,'Enable','off');
        handles.data = handles.original_data;
        display_data(handles);
    end
end
```

Listing 5.9: Display Original Unprocessed Signal Data

This function simply sets the correct state of the toolbar buttons and if transformation

processes exist, it disables the 'Processing' button and copies `original_data` to `data` before calling the `display_data` function to plot the unprocessed data.

While the unprocessed signal data is being displayed, the representations of the processed signal do not exist. Therefore when 'View Processed Data' is clicked, the `run_processes` function is used to calculate the processed signal again before it can be displayed. This is shown in listing 5.10 within the function `tb_viewProcCallback`, which is the callback of the toolbar button.

```
function tb_viewProcCallback(hObject, eventdata)
    handles = guidata(gcf);
    if strcmp(get(handles.tb_viewOrig,'State'),'off')
        set(handles.tb_viewProc,'State','on');
    else
        set(handles.tb_viewOrig,'State','off');
    end
    set(handles.process_button,'Enable','on');
    handles.original_data = handles.data;
    handles = run_processes(handles,handles.processes.num_proc);
    display_data(handles);
end
```

Listing 5.10: Display Processed Signal Data

Note how the `data` structure is copied into `original_data` before the processed data is recalculated. This is because the 'Info' GUI window allows the user to adjust the pitch analysis information (section 4.6.4), which would change the original unprocessed data representation. The pitch analysis adjustment in the 'Info' window is not made available to the user when the processed signal data is being viewed, since the analysis parameters can be changed by the transformation processes.

5.1.7 Recalculating Signal Properties When Analysis Settings Change

The `analyse_audio` function has already been discussed in depth, but it has not been stated that the `run_processes` function is called before the `loaded` function. This allows

the data to be reanalysed even when transformation processes are applied.

Two of the transformation processes, filtering (section 5.2) and resizing (section 5.9) of the spectrum data, use the signal analysis settings when calculating their process parameters. Therefore when the analysis settings of the signal data are altered, these processes need a means to recalculate their parameters appropriately. The two functions `calc_filter_properties` and `calc_resize_properties` were written to adjust the necessary parameters of the filter and resize signal transformations when analysis settings are changed. The details of each of these functions will be covered in the respective process sections, however their integration into the software tool is discussed here.

Before `run_processes` is called, a loop is set up so that parameters are recalculated for any filter or resize processes in the `process` array. Listing 5.11 shows an excerpt from the end of the `analyse_audio` function to clarify the sequence of instructions.

```
% calculate images
handles = calc_image(handles);
% create the spectrum images
handles = spec2D_init(handles);
% store data as original also
handles.original_data = handles.data;
% get processed data

% must recalculate the process properties according to the new data, if
% necessary
for n = 1:handles.processes.num_proc
    switch handles.processes.process(n).type
        case handles.proc_names{2} % filter
            handles = calc_filter_properties(handles,n,false);
        case handles.proc_names{15} % resize
            handles = calc_resize_properties(handles,n);
    end
end
handles = run_processes(handles,handles.processes.num_proc);
%   complete loading
loaded(hObject,handles);
```

Listing 5.11: Recalculating Process Parameters and Running Processes at the End of `analyse_audio`

The resize process, and also the rotation process (section 5.4), can alter the signal analysis parameters during their processing. Therefore any time either of these processes is run, any subsequent resize or filter processes must again recalculate their parameters according to the new analysis settings. Within the `switch...case...` statement of `run_processes`, the resize and rotate cases both contain similar code to `analyse_audio` allowing the parameters of filter and resize processes to be recalculated, though only subsequent processes need to be considered, as shown in listing 5.12.

```
if n<handles.processes.num_proc
    for m = n+1:handles.processes.num_proc
        switch handles.processes.process(m).type
            case handles.proc_names{2}
                handles = calc_filter_properties(handles,m,false);
            case handles.proc_names{15}
                handles = calc_resize_properties(handles,m);
        end
    end
end
```

Listing 5.12: Recalculating Process Parameters in `run_processes`

The index `n` identifies the current process in the list i.e. a resize or rotate process, and the incrementing index `m` allows subsequent processes in the `process` array to be examined, since only these will be affected by the change in analysis parameters. When a resize or rotate transformation is removed from the `process` array, the parameters of subsequent filter or resize processes again need recalculating, since the analysis settings are no longer being changed. This occurs in the `cmenu_clicked` callback function within `proc_popup_callbacks.m`.

5.2 2D Fourier Spectrum Filtering

The concept of filtering in the 2D frequency domain was one of the most immediately obvious and exciting transformation processes. It not only offers filtering of the audible frequency range as with conventional time and frequency domain filters, but also in the rhythmic frequency range, allowing the sub-sonic rhythmic variations in the signal to be manipulated.

5.2.1 Filter Parameters

The 2D spectral filter that was developed in the software tool provides many different options. Either frequency dimension can be filtered using one of four common filter types (low-pass, high-pass, band-pass and band-stop) to either cut stop-band frequencies or boost pass-band frequencies. The filter can have an ideal 'brick-wall' frequency response or the frequency response of a Butterworth architecture of any order chosen by the user. Additionally DC content of the spectrum can be retained even if it is in the stop-band. The variables that correspond to the filter options are given in table 5.1, along with their default values that are applied when the filter process is created, using the `create_filter` function. These variables will be stored in the structure within the `process` array that corresponds to the filter process.

Aside from these options, the parameters that define the filter process are the cutoff frequency and if the filter is a band-pass or band-stop, the bandwidth of the filter. The cutoff frequency actually corresponds to the centre frequency of a band-pass or band-stop filter. The properties of the 2D Fourier spectrum have to be analysed to calculate the range of values that the cutoff/centre frequency and bandwidth of the filter can hold. This is done using the function `calc_filter_properties`, which is called when the filter is created in `create_filter` and also whenever the properties of the 2D Fourier spectrum are changed, see section 5.1.7. The Boolean parameter `new` is given as an argument in `calc_filter_properties` to identify between the two situations. If `new` is true, then the function has been called from `create_filter` and the default cutoff and bandwidth of the filter need to be set. The cutoff/centre frequency (`cutoff`) is set to the maximum rhythmic frequency represented by the 2D Fourier spectrum of any signal frame (the filter is in

rhythmic frequency mode by default). The bandwidth (`bw`) is set to two times the minimum frequency interval between adjacent bins on the rhythmic axis of the 2D spectrum of any signal frame.

Variable	Data Type	Default Value	Description
`type`	String	'Filter'	The name of the process, obtained from the `proc_names` array. Used to identify the process as a filter.
`bypass`	Boolean	`false`	Determines whether or not the filter process should be bypassed when the chain of transformations is run.
`ftype`	String	'LP'	The filter type: 'LP', 'HP', 'BP' or 'BS'.
`rhythmic_mode`	Boolean	`true`	Determines which axis of frequency is filtered. The rhythmic frequencies are filtered when `true` and audible frequencies when `false`.
`keepDC`	Integer	0	Indicates what DC component, if any, should be preserved. Either no data is preserved (0), the single point that is DC in audible and rhythmic frequency (1), all audible frequencies with DC rhythmic frequency (2) or all rhythmic frequencies with DC audible frequency (3).
`cut`	Boolean	`true`	Determines whether filter cuts the stop-band or boosts the pass-band.
`ideal`	Boolean	`true`	Determines whether the filter's frequency response is ideal or Butterworth.
`order`	Integer	2	The order of Butterworth filter used to define the frequency response when `ideal = false`.

Table 5.1: 2D Spectral Filter Parameters Set in `create_filter`

5.2.2 Calculating Filter Properties Based On The 2D Fourier Spectrum

The function `calc_filter_properties` is used to determine a series of filter properties that help to define the range of values that `cutoff` and `bw` can take when being adjusted by the user. Table 5.2 shows the variables obtained by this function, which are stored in the `process` structure.

Variable	Description
`rhyth_freq`	2D array of rhythmic frequencies represented by the 2D Fourier spectrum of each signal frame.
`aud_freq`	2D array of audible frequencies represented by the 2D Fourier spectrum of each signal frame.
`max_rhyth_frame`	The frame number of the 2D spectrum that represents the highest maximum rhythmic frequency.
`max_aud_frame`	The frame number of the 2D spectrum that has the maximum number of points in the audible dimension.
`max_cut`	The maximum allowed value of `cutoff`, the filter's cutoff frequency, in the currently selected frequency axis and also the maximum filter bandwidth, `bw`. This variable is determined by the maximum frequency in the appropriate axis represented by the 2D spectrum of any signal frame.
`min_freq_step`	The minimum frequency interval between adjacent bins on the currently selected frequency axis of the 2D spectrum of any signal frame. This is the minimum value of the `bw` variable.

Table 5.2: Filter Data Obtained From The 2D Fourier Spectrum

The `calc_filter_properties` function determines the frequency arrays corresponding to the centre of each frequency bin on both axes of the 2D spectrum and stores them in `rhyth_freq` and `aud_freq` respectively. When the signal analysis is in timbral mode this needs to be done for every frame, since each has its own 2D spectrum, resulting in 2D arrays. As these frequency arrays are calculated, the `max_rhyth_frame` and `max_aud_frame`

parameters are updated to contain the index of the signal frame containing the required rhythmic and audible frequency arrays respectively.

The value of the filter's `cutoff` variable can range from 0Hz to the maximum frequency represented in the 2D spectrum of any of the signal frames, in the frequency axis indicated by `rhythmic_mode`. This maximum value is held in the `max_cut` variable.

The minimum value of the filter's `bw` variable is `min_freq_step`, which is the minimum interval between frequency bins of the signal's 2D spectra, on the axis indicated by `rhythmic_mode`.

If the `calc_filter_properties` function has been called as a result of altered analysis settings then the filter's `cutoff` and `bw` variables are checked and if out of range according to the new settings, they are set to the appropriate range limit. This prevents an error from occuring when the filter process is run.

It is worth noting that in the processing stage, signals in rhythmic analysis mode are considered by the software to have only one frame, since there is only one 2D spectrum to represent the signal. This is as opposed to the definition of a frame as a row of the raster image given in section 4.5.2. This was a design fault that was only realised once the analysis phase of the project was completed. At this point it seemed the original definition of signal frames was too deeply implemented into the software. To work around the problem, most processing operations require the commands in listing 5.13. A function containing this code would then use the `nframes` variable in replacement of `num_frames`.

```
if strcmp(handles.data.analysis_settings.analysis_mode,'rhythmic')
    nframes = 1;
else
    nframes = handles.data.analysis.num_frames;
end
```

Listing 5.13: Correcting the Number of Frames Parameter in Processing Functions

5.2.3 Adjusting Filter Parameters

The parameters of a 2D spectral filter can be adjusted using the `adjust_filter` function, which creates the GUI shown in figure 5.3. This GUI window allows the user to set all

of the filter options as desired and specify the cutoff frequency of the filter and also the bandwidth if it is required. It also displays the filter's frequency response as an image, to give the user a better impression of how the filter will affect the 2D Fourier spectrum of the signal.

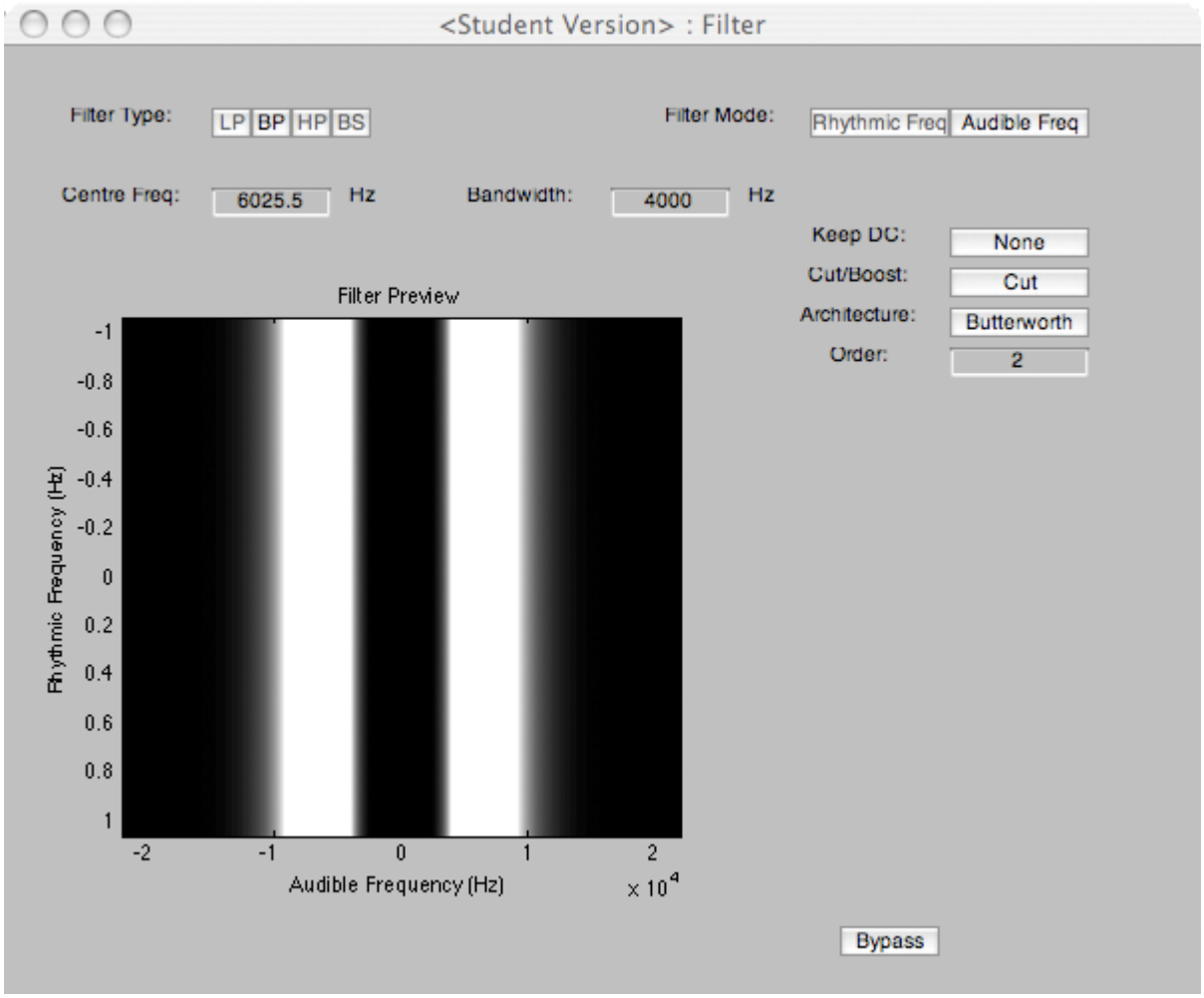


Figure 5.3: 2D Spectral Filter GUI

The callback functions of the GUI's button and text edit objects update the appropriate filter parameters in the `cur_process` structure (section 5.1.4) and then call the function `filter_settings_changed`. This function checks whether the cutoff frequency is still in the correct range, between `max_cut` and 0. If the filter type (`ftype`) is set to band-pass (`BP`) or band-stop (`BS`) then the bandwidth is also limited to its range, between `min_freq_step` and `max_cut`. The text label next to the `cutoff` value is set to 'Cutoff Freq' for a low-

pass or high-pass filter and 'Centre Freq' for a band-pass or band-stop filter. The text edit objects for cutoff and bandwidth are updated with the current `cutoff` and `bw` values, then the image plot of the filter's frequency response is calculated and displayed. Finally the `process_changed` variable is set to true to indicate the filter parameters have been adjusted (section 5.1.5) before the `handles` structure is stored in the filter figure's GUI data.

The filter image is created and displayed using the `create_filter_image` function. The filter's frequency response is calculated using the function `calc_filt`, which is also used to run the filter process and so is described in section 5.2.4. The filter image is created at the size of the 2D spectrum of the frame indicated by either `max_rhyth_frame` or `max_aud_frame` depending on the value of the `rhythmic_mode` variable. This frame index is also used to obtain the frequency array corresponding to the frame, allowing the image to be created. The image is returned from `calc_filt` and displayed on the axes within the filter figure using Matlab's `image` function. The rhythmic and audible frequency arrays are obtained from the `rhyth_freq` and `aud_freq` variables and displayed along the axes of the plot.

5.2.4 Running the 2D Spectral Filter

The filter process is run using the function `process_filter`, which creates an array of the filter's 2D frequency response and multiplies it (element-wise) by the magnitude component for each frame of 2D Fourier data. The processed 2D Fourier data array, `FT`, is then calculated by combining the new magnitude component with the phase component to obtain a complex representation by the equations:

$$\Re\left(\mathtt{FT}\right) = \left|\mathtt{FT}\right| \cos\left(\angle\left(\mathtt{FT}\right)\right) \tag{5.1}$$

$$\Im\left(\mathtt{FT}\right) = \left|\mathtt{FT}\right| \sin\left(\angle\left(\mathtt{FT}\right)\right) \tag{5.2}$$

The `calc_filt` function requires three input arguments:

- `process_data` - The filter process' structure, as stored in the `process` array, providing the parameters and settings of the filter.
- `filt_image` - An empty 2D array of the correct size to take the filter's frequency response data.

- `freq_array` - A 1D array of the frequency values at each point along the axis of filtering.

In order to make `calc_filt` generic for any filter, the filter function is always defined along the columns of the `filt_image` array, which is then rotated by 90° if the `rhythmic_mode` variable is false. The dimensions of the `FT` array are therefore reversed in `process_filter`, to create the `filt_image` array for an audible frequency filter.

Within `calc_filt`, a `switch...case...` statement is used to determine the required filter type according to the value of the `ftype` variable. Within each filter type block, there is the code for both the ideal and Butterworth options. The general concept for the ideal filtering is to determine whether each frequency within `freq_array` is within the pass-band or the stop-band and that row of `filt_image` is set to 1 or 0 accordingly. The implementation of the ideal band-pass filter function is shown in listing 5.14, the ideal band-stop filter implementation is identical apart from the 1 and 0 being swapped. The `centre` variable is the index of the 0 Hz point within the frequency array, this is the same for all ideal filter implementations.

```
centre = ceil(length(freq_array)/2);
for n = 1:length(freq_array)
    if n < centre
        freq_pt = -freq_array(n);
    else
        freq_pt = freq_array(n);
    end
    if freq_pt >= (process_data.cutoff - process_data.bw/2) && ...
        freq_pt <= (process_data.cutoff + process_data.bw/2)
        filt_image(n,:) = 1;
    else
        filt_image(n,:) = 0;
    end
end
```

Listing 5.14: Implementation of an Ideal Band-Pass Filter

The ideal low-pass and high-pass frequency responses are created by inserting arrays of ones into the `filt_image` array (initially all zeros) rather than calculating one row at a

time. The ideal low-pass implementation is shown in listing 5.15. Matlab's `find` function is used to obtain the index of the last frequency point in `freq_array` that is lower than the cutoff frequency, determining the size of the array of ones that is inserted into `freq_image`, centred around 0 Hz. The implementation of the high-pass filter is similar although two arrays of ones are required, placed at the top and bottom of the `filter_image`.

```
ind_cut = find(freq_array<=process_data.cutoff,1,'last');
num_steps = ind_cut - centre;
filt_image((centre-num_steps):ind_cut,:) = ones((2*num_steps+1),...
    size(filt_image,2));
```

Listing 5.15: Implementation of an Ideal Low-Pass Filter

The implementation of the Butterworth filtering is quite different to ideal filtering. The filter function is defined by an equation that takes in the `freq_array` and an equally sized array `cut_array` with every value set to `cutoff`. The result is a 1D array of the filter's response for each frequency in `freq_array`, that can then be inserted into every column of the `filt_image` array. The function for the Butterworth low-pass filter is given by equation 5.3, the high-pass filter by equation 5.4, and the band-pass and band-stop filters by equations 5.5 and 5.6 respectively.

$$H(u) = \frac{1}{1 + [F(u)/F_0]^{2n}} \tag{5.3}$$

$$H(u) = \frac{1}{1 + [F_0/F(u)]^{2n}} \tag{5.4}$$

$$H(u) = 1 - \frac{1}{1 + \left[\dfrac{F(u) * W}{F(u)^2 - F_0^2}\right]} \tag{5.5}$$

$$H(u) = \frac{1}{1 + \left[\dfrac{F(u) * W}{F(u)^2 - F_0^2}\right]} \tag{5.6}$$

Where n is the filter order (`order`), F_0 is the cutoff/centre frequency (`cutoff`), F is the frequency array (`freq_array`), W is the filter bandwidth (`bw`), H is the filter's frequency response and u is the index within the arrays.

Once the frequency response has been obtained, the filter's `keepDC` variable is inspected

and if required, the appropriate DC points are set to 1 in `filt_image`, as shown in listing 5.16. This functionality was incorporated to mainly for analysis purposes, to investigate the effects of retaining the DC rhythmic frequency row during, for example, high-pass filtering.

```
switch process_data.keepDC
    case 1
        mid_row = ceil(size(filt_image,1)/2);
        mid_col = ceil(size(filt_image,2)/2);
        filt_image(mid_row,mid_col) = 1;
    case 2
        mid_row = ceil(size(filt_image,1)/2);
        filt_image(mid_row,:) = ones(1,size(filt_image,2));
    case 3
        mid_col = ceil(size(filt_image,2)/2);
        filt_image(:,mid_col) = ones(size(filt_image,1),1);
end
```

Listing 5.16: Retaining the DC Component When Filtering

If `cut` is set to false, the filter frequency response is adjusted when returned from `calc_filt` to `process` filter, to give the pass-band frequencies a 20 dB boost whilst leaving stop-band frequencies unaltered. This is achieved by the following equation:

$$\texttt{filt\_image} = (\texttt{filt\_image} + 1) * 10 \tag{5.7}$$

Such a significant amplitude increase is applied so that the effects of boosting frequency may be easily observed. In a revised implementation, the filter would be implemented with a variable amplitude increase.

5.3 Magnitude Thresholding with the 2D Fourier Spectrum

Magnitude thresholding allows spectral components to be separated according to their magnitude by setting a threshold value below/above which 2D spectrum points are re-

moved. This process serves as a useful analysis tool for determining the level of influence that the strongest/weakest spectral components have in defining the audio signal.

5.3.1 Thresholding Parameters

The magnitude thresholding process features a threshold parameter defined as a percentage of the spectrum's maximum magnitude value, in order to make it intuitive to operate and easy to implement. The process can either be applied to individual points within the spectrum or the rows of rhythmic frequency or columns of audible frequency, using the sum of their magnitude values. It also provides the option of removing components that are either above or below the threshold value. The variables that define these parameters are given in table 5.3, along with their initial default values as defined in the function `create_thresh`. These variables are stored in the data structure corresponding to a thresholding process within the `process` array.

5.3.2 Adjusting the Thresholding Parameters

The `adjust_thresh` function allows the parameters of a thresholding process to be viewed and adjusted by creating a GUI window as shown in figure 5.4. This window allows the user to set the process options using push buttons defined as button groups in code. The threshold value is set using a slider object with the value range [0 1] and the text next to the slider displays this value as a percentage (rounded to the nearest integer). When any of the options are adjusted, or the threshold slider moved, the object callback functions set the appropriate variables in the `cur_process` structure and the `process_changed` variable is set to true to indicate that the settings have been adjusted.

Variable	Data Type	Default Value	Description
`type`	String	`proc_names{3}` = 'Threshold'	The name of the process, obtained from the `proc_names` array. Used to identify the process as magnitude thresholding.
`bypass`	Boolean	`false`	Determines whether or not the thresholding process should be bypassed when the chain of transformations is run.
`ttype`	String	'rhythmic_freq'	The type of frequency component that is processed. Either single spectrum points with defined rhythmic and audible frequency ('single_point'), spectrum rows with defined rhythmic frequency only ('rhythmic_freq') or spectrum columns with defined audible frequency only ('audible_freq').
`thresh`	Double	0	The magnitude threshold value given as a proportion of the maximum magnitude with the range [0 1].
`below_thresh`	Boolean	`true`	Determines whether the threshold removes components below the threshold value (if `true`) or above the threshold value (if `false`).

Table 5.3: 2D Magnitude Threshold Parameters Set in `create_thresh`

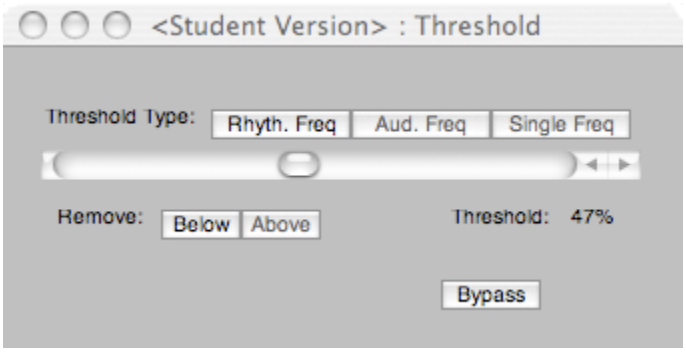


Figure 5.4: 2D Magnitude Thresholding GUI

5.3.3 Running the Magnitude Thresholding

The magnitude thresholding process is run in `process_thresh`, which calculates the new `FT` data frame-by-frame. For each frame, the magnitude and phase components are first extracted from the `FT` array using `fftshift`, `abs` and `angle`. Then a `switch...case...` statement is used to determine the type of thresholding to be applied, given by the `ttype` variable. For each thresholding type the maximum magnitude must be calculated using Matlab's `max` function. When thresholding by only rhythmic or audible frequency the sum of magnitudes of each the row/column is required first. The actual magnitude value of the threshold can then be calculated from this maximum and the value of `thresh`. The magnitude value of each component is compared with the threshold value and set to zero if above/below depending on the `below_thresh` setting. The thresholding of rhythmic frequency is shown in listing 5.17 as an example.

```
totals = sum(rot90(mag));
max_tot = max(totals);
thresh_val = handles.processes.process(process_num).thresh*max_tot;
for row = 1:size(mag,1)
    if handles.processes.process(process_num).below_thresh
        if totals(row) < thresh_val
            mag(row,:) = zeros(1,size(mag,2));
        end
    else
        if totals(row) > thresh_val
```

```
                mag(row,:) = zeros(1,size(mag,2));
            end
        end
    end
```

Listing 5.17: Thresholding Rhythmic Frequencies

Once the new magnitude component has been determined, it is recombined with the phase to produce the new complex `FT` array using equations 5.1 and 5.2, followed by the `rev_fftshift` function.

5.4 2D Fourier Spectrum Rotation

This process lets the user rotate the 2D Fourier spectrum in 90° increments. When rotations of 90° or 270° are applied the frequency axes of the 2D spectrum are swapped. The original rhythmic frequency bins can be redefined as audible frequencies in the range [0 $F_s/2$] and the original audible frequencies would then define a new range of rhythmic frequency variation in the vertical axis.

5.4.1 Spectral Rotation Parameters

There is only one parameter for the rotation process, apart from the `type` and `bypass` variables used by all adjustable processes. This is the `rot_ind` parameter which describes the rotation of the spectrum, r, by the equation:

$$r = \texttt{rot\_ind} * 90^\circ \tag{5.8}$$

The default value of `rot_ind` is 1, which is defined when the rotation process is created in `create_rot_spec`.

5.4.2 Adjusting the Rotation

The `adjust_rot_spec` function creates a GUI window, as shown in figure 5.5, that allows the amount of rotation to be set. It simply presents a drop-down menu with the three

options: '90 degrees', '180 degrees' and '270 degrees'. The index of the menu items is equal to the value of `rot_ind` required for the displayed rotation, therefore the selected item index can be stored in `rot_ind` within the menu's callback function.

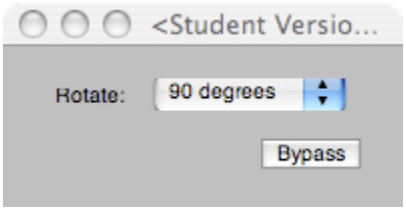


Figure 5.5: 2D Magnitude Thresholding GUI

When the signal has been analysed in pitch-synchronised rhythmic mode, the spectrum can only be rotated by 180°. This is because the conversion from raster image to audio is a complex process in this mode, based on the calculated pitch of each raster image row (section 4.9.4). If the frequency axes were swapped due to rotation then the current resynthesis process would not work. It was decided that it wasn't appropriate to develop the resynthesis process to compensate for rotation adjustments, since pitch-synchronous rhythmic analysis had already been deemed of little use in the project. The drop-down menu therefore only presents one option, '180 degrees' and the value of `rot_spec` is forced to 2 when the rotation process is performed.

5.4.3 Running the Spectral Rotation Process

The `process_rot_spec` function performs the rotation process upon each frame of 2D spectral data in the `FT`. The process very simply rotates the `FT` array in 90° increments `rot_ind` times, using Matlab's `rot90` function. However, when a rotation of 90° or 270° is applied, the analysis settings of the signal have to be appropriately altered to reflect the swapped dimensions of the 2D representation. The code that adjusts the analysis parameters of the data is shown in listing 5.18.

```
% if rot_ind is 1 or 3 then analysis data has to change
if mod(handles.processes.process(proc_num).rot_ind,2)
    temp = handles.data.spec2D_settings.height_pad;
    handles.data.spec2D_settings.height_pad = ...
```

```
        handles.data.spec2D_settings.width_pad;
    handles.data.spec2D_settings.width_pad = temp;
    if strcmp(handles.data.analysis_settings.analysis_mode,'timbral')
        for frame = 1:handles.data.analysis.num_frames
            handles.data.analysis.imwidth(frame) = size(...
                handles.data.FT{frame},2) - ...
                handles.data.spec2D_settings.width_pad(frame);
        end
    else
        % if rhythmic mode and not synced
        handles.data.analysis.imwidth = handles.data.analysis.num_frames;
        temp = handles.data.analysis.frame_size;
        handles.data.analysis.frame_size = handles.data.analysis.num_frames;
        handles.data.analysis.num_frames = temp;
        handles.data.analysis_settings.frame_size_secs = handles.data.
            analysis.frame_size/...
            handles.data.audio_settings.Fs;
        handles.data.analysis_settings.tempo = (240*...
            handles.data.analysis_settings.tempo_div_num)/...
            (handles.data.analysis_settings.frame_size_secs*...
            handles.data.analysis_settings.tempo_div_den);
    end
end
```

Listing 5.18: Correcting Analysis Settings After Rotation in `process_rot_spec`

The `width_pad` and `height_pad` variables need to be swapped so that the correct amount of zero padding can be removed from each dimension when the raster image is obtained from the spectrum. Any analysis settings corresponding to the dimensions of the raster image then need to be recalculated. In timbral analysis mode, the width of the raster images (`imwidth` array) is obtained using the `FT` array sizes and the new values of `width_pad`. In rhythmic mode (without pitch-synchronisation only), the `imwidth` and `frame_size` variables, which are both the same, need to be swapped with `num_frames`, since these parameters correspond to the dimensions of the raster image.

5.5 2D Fourier Spectrum Row Shift

This process allows the rows of rhythmic frequency content to be shifted within the 2D Fourier spectrum, setting them to a new rhythmic frequency value. This process is a useful tool to aid the understanding of rhythmic frequency content and provides interesting perceptual effects when transforming audio.

5.5.1 Row Shift Parameters

The row shift process uses the integer parameter, `shift`, to define the number of rows by which the spectrum data is shifted. The rows are shifted away from 0 Hz for a positive value of `shift`, so positive and negative frequency rows are shifted in opposite directions, to maintain the symmetry of the 2D spectrum. The `shift` parameter can take any integer value however the amount of rows actually shifted is given by the equation:

$$\texttt{row\_shift} = \texttt{mod}(\texttt{shift}, \texttt{ceil}(M/2)) \tag{5.9}$$

Where M is the height of the 2D spectrum, `ceil` is the Matlab function that rounds the input towards infinity and `mod` is the Matlab function that returns the modulus after division. This operation is performed in the `process_row_shift` function. It means that a `shift` value of any integer, including negatives, can be converted to a value in the range [0 `ceil`$(M/2)$].

The row shift process also has two Boolean option parameters, `wrap` and `remove`, which together define the three different modes of operation. When `wrap` is true, any rhythmic frequency rows that are shifted beyond the end of the spectrum are wrapped back around starting from the 0 Hz row, meaning that every row in the spectrum contains new data. If `wrap` is false, then `remove` determines what data is stored in the low frequency rows that are not overwritten (this will match the value in `shift` after equation 5.9). If `remove` is true then these rows will be empty, containing only zeros, otherwise the original row data is retained.

With these options defined, it can now be shown that a row shift greater than `ceil`$(M/2)$ is never required. When the rows are wrapped around, the shifting is periodic about `ceil`$(M/2)$ and if the rows are not wrapped round, then a shift of more than `ceil`$(M/2)$

would lead to either an empty spectrum or a spectrum identical to the original.

5.5.2 Adjusting the Row Shift Process

The `adjust_row_shift` function allows the row shift options to be observed and adjusted by creating the GUI shown in figure 5.6. There is a single push button component that toggles between the three options described in section 5.5.1, displaying the following one of the following three strings according to the option selected: 'Wrap Rows Around', 'Leave Orig. Rows' and 'Remove Orig. Rows'. The values of the `wrap` and `remove` variables are set appropriately for each option in the button's callback function.

The text edit object's callback simply rounds the numerical input to the nearest integer and stores it in `shift`. The text edit object's string is then updated with the rounded value.

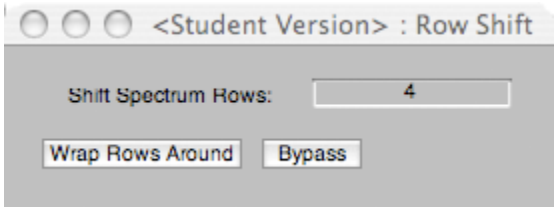


Figure 5.6: Rhythmic Frequency Row Shift GUI

5.5.3 Running the Row Shift Process

As with the other `process_*` functions, `process_row_shift` performs its operation on each frame's 2D spectrum separately. Listing 5.19 shows the code used to calculate the shifted `FT` array for each frame of data. Equation 5.9 must be used to calculate the actual number of rows to shift for each frame, since the height of the spectrum is variable. The function uses two arrays `oldFT` and `newFT` to hold the Fourier data arrays of the input and output respectively. If the `remove` variable is set to true, then `newFT` is initialised as a copy of `oldFT`, otherwise it is initially filled with zeros. The two halves of the spectrum, with positive and negative rhythmic frequency are calculated separately.

```
width = size(handles.data.FT{frame},2);
```

```
height = size(handles.data.FT{frame},1);
oldFT = fftshift(handles.data.FT{frame});
if handles.processes.process(proc_num).remove
    newFT = zeros(height,width);
else
    newFT = oldFT;
end
% shift is always < floor(height/2);
% also quadrant dims
middle = ceil(height/2);
width_middle = ceil(width/2);
row_shift = mod(handles.processes.process(proc_num).shift,middle);

% top half of spec
end_top_n = middle-row_shift;
newFT(1 : end_top_n,:) = oldFT((row_shift+1) : middle,:);
if handles.processes.process(proc_num).wrap && row_shift ~= 0
    newFT((middle-row_shift+1) : (middle-1),:) = oldFT(1:(row_shift-1),:);
    newFT(middle,1:(width_middle-1)) = oldFT(row_shift,1:(width_middle-1));
end
% bottom half of spec
end_bottom_n = middle+row_shift;
newFT(end_bottom_n:height,:) = oldFT(middle : (height-row_shift),:);
if handles.processes.process(proc_num).wrap && row_shift ~= 0
    newFT((middle+1) : (middle+row_shift-1),:) = ...
        oldFT((height-row_shift+2) : height,:);
    newFT(middle,width_middle:width) = oldFT((height-row_shift+1),...
        width_middle:width);
end
handles.data.FT{frame} = rev_fftshift(newFT);
```

Listing 5.19: Shifting Rows Of The 2D Spectrum in `process_row_shift`

The data is shifted using sub-arrays of the `oldFT` array to increase the process operation speed. Matlab processes array mathematics much faster than program loops, since it is an interpreted programming language (section 2.6). Note how, when the `wrap` variable is true, the new 0 Hz row, at the index given by `middle`, is defined in two halves. The negative audible frequencies are obtained from the row that originally had negative rhythmic frequency and the positive audible frequencies are obtained from the row that originally

had positive rhythmic frequency. This ensures that the symmetry of the 0 Hz rhythmic frequency row is maintained (see section 4.3.2).

5.6 2D Fourier Spectrum Column Shift

The column shift process allows the columns of audible frequency content to be shifted within the 2D Fourier spectrum, giving them a new audible frequency value. This process can be used to adjust the audible frequency content of an audio signal whilst maintaining the same rhythmic structure.

The parameters and implementation of the column shift process are identical to those of the row shift process (section 5.5) apart from the frequency axis on which the process operates and therefore these details are omitted.

5.7 Pitch Shifting with the 2D Fourier Spectrum

Conventional 1D frequency domain pitch shifting is obtained by scaling the frequency values of each component [26]. The same technique has been implemented in the 2D Fourier spectrum. The audible frequency bin values are rescaled by a linear factor according to the desired pitch change, and the data is then resampled to get this scaled data at the original audible frequency values.

5.7.1 Pitch Shift Parameters

The pitch shift process only has one parameter, `shift`, which gives the required pitch shift in semitones. The value of `shift` can be any integer, positive or negative, and the process will rescale the audible frequency data by the linear factor corresponding to this shift. The initial setting for `shift` is 0, defined in the function `create_pitch_change` which initialises the pitch shift process.

5.7.2 Adjusting the Pitch Shift Process

The function `adjust_pitch_change` allows the user to set the value of the `shift` parameter using a text edit object in the GUI shown in figure 5.7. The callback for this text edit object rounds the numerical input and stores the value in `shift`, and corrects the text edit string.

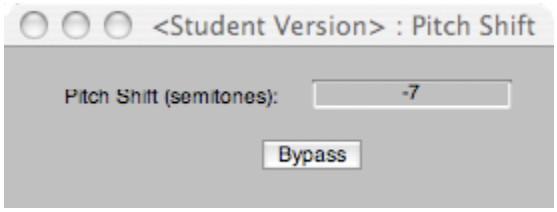


Figure 5.7: Pitch Shift GUI

5.7.3 Running the Pitch Shift Process

The function `proces_pitch_change` performs the operations shown in listing 5.20 for each 2D spectrum frame. The linear scaling factor (`change` variable) for the desired pitch change is calculated from the `shift` value using the following equation:

$$\texttt{change} = 1/2^{\texttt{shift}/12} \tag{5.10}$$

The data for the pitch-shifted `FT` array is calculated from the original `FT` array by resampling the data rows. Matlab's `interp1` function to perform cubic spline interpolation to resample the data. The arrays `x` and `xi` are calculated to give the input and output indices of the row signal for the interpolation. The sample point arrays and the original row of the `FT` array are input to the `interp1` function, which returns the pitch shifted row. The 2D Fourier data must be quadrant-shifted before and after the resampling to ensure that the data is correctly processed The `interp1` function is set to insert zeros when the output sample points are outside the range of the input function. This was chosen as opposed to extrapolation, since the audible frequency envelope is likely to be too complex to extrapolate accurately for an upwards pitch shift, bearing in mind that shifting the pitch down an octave would mean that half of the audible frequency information would have to be

calculated by extrapolation.

```
% perform pitch shift on FT once fftshifted
width = size(handles.data.FT{frame},2);
height = size(handles.data.FT{frame},1);
oldFT = fftshift(handles.data.FT{frame});
newFT = zeros(width,height);
x = linspace(-1,1,width);
xi = linspace(-change,...
    change,width);
row = 1:height;
newFT(:,row) = interp1(x,rot90(oldFT(row,:)),xi,'spline',0);
handles.data.FT{frame} = rev_fftshift(rot90(rot90(rot90(newFT))));
```

Listing 5.20: Pitch Shifting Using the 2D Fourier Spectrum in `process_pitch_change`

5.7.4 Fixed Pitch Shifting Processes

As well as the 'Pitch Shift' process, there are also two 'fixed' pitch shifting processes, 'Up Octave' and 'Down Octave'. As section 5.1.4 stated, fixed processes do not require a `create_*` or `adjust_*` function, since they do not offer any adjustable parameters. The function `process_octave` defines the operation of these two processes, with the Boolean input argument `up` determining the direction of octave shift. These two processes were written to investigate the quality of the pitch shifting process described above, without the effects of interpolation error. A pitch increase of an octave can easily be obtained by inserting a zero between each of the audible frequency points, using only the points up to half the maximum audible frequency ($F_s/4$). A pitch decrease of an octave can easily be obtained by removing every other audible frequency point and padding the quadrant-shifted Fourier data array to the original size with an equal number of zeros on either side. Listing 5.21 shows the operations of the `process_octave` function for each 2D spectrum frame. Each pitch change is performed as an array operation in one line, using the `up` variable to determine the appropriate equation.

```
width = size(handles.data.FT{frame},2);
height = size(handles.data.FT{frame},1);
```

```
        oldFT = fftshift(handles.data.FT{frame});
        newFT = zeros(height,width);
        start = ceil(width/4);
        col = 1:ceil(width/2);
        if up
            newFT(:,(2*col -1)) = oldFT(:,col+start);
        else
            newFT(:,col+start) = oldFT(:,(2*col -1));
        end
        handles.data.FT{frame} = rev_fftshift(newFT);
```

Listing 5.21: Octave Pitch Shifting Without Interpolation in `process_octave`

5.8 Rhythmic Frequency Range Compression/Expansion

Rhythmic frequency compression and expansion processing is performed by resampling the columns of the 2D Fourier domain data to alter the range of rhythmic frequency. Its operation is similar to the pitch shifting process in section 5.7, but for the rhythmic frequency axis rather than audible frequency. Figure 5.8 attempts to visually describe the process, demonstrating how the range of the original rhythmic frequencies can be stretched beyond the scope of the 2D Fourier signal representation or compressed to less than the original range. This process is referred to within the software tool using the short name 'Stretch Rhythm' and will be commonly referred to as the rhythmic frequency stretching process in this section.

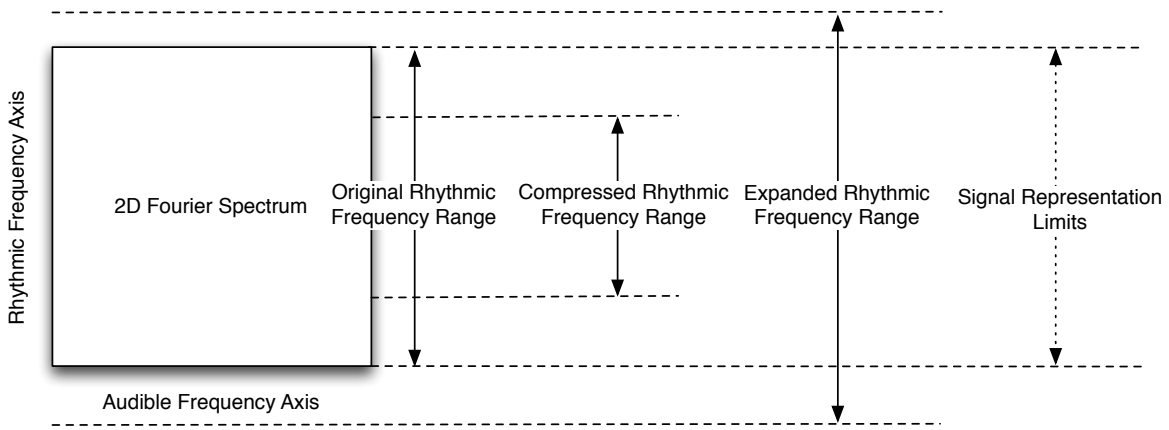


Figure 5.8: Rhythmic Frequency Range Compression/Expansion

5.8.1 Rhythmic Frequency Stretching Parameters

The function `create_spec_stretch` is used to initialise this rhythmic frequency stretching process. As with the pitch shifting process, there is only one adjustable parameter which corresponds to the linear factor by which the frequency range is adjusted. However in rhythmic frequency stretching this variable, named `amount`, is the precise value of range adjustment; there is no intermediate equation before running the process. The `amount` variable is initially set to 0 in `create_spec_stretch`, it can be set to any positive number within the range of Matlab's double precision floating point data type.

5.8.2 Adjusting the Rhythmic Frequency Stretching Process

The GUI window shown in figure 5.9 is created using `adjust_spec_stretch`. It allows the user to set the value of the rhythmic stretching process' `amount` parameter using a text edit object. If a negative value is entered then its absolute value is stored in `amount` and the text edit object's string is corrected accordingly.

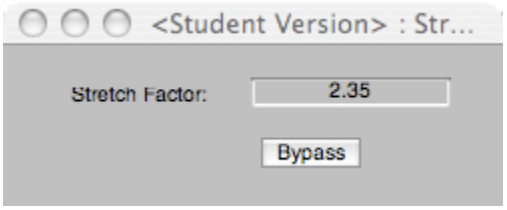


Figure 5.9: Rhythmic Frequency Range Stretching GUI

5.8.3 Running the Rhythmic Frequency Stretching Process

Rhythmic frequency range compression/expansion is performed by resampling the columns of the 2D Fourier data array, adjusting the range of frequencies that the original data points represent. This process is performed in the `process_spec_stretch` function, processing each data frame separately. The operations performed on the 2D Fourier data each frame are shown in listing 5.22. The `amount` variable is used to calculate `xi`, the array of output sample indices for the `interp1` function.

```
width = size(handles.data.FT{frame},2);
height = size(handles.data.FT{frame},1);
oldFT = fftshift(handles.data.FT{frame});
newFT = zeros(height,width);
x = linspace(-1,1,height);
xi = linspace(-1/handles.processes.process(proc_num).amount,...
    1/handles.processes.process(proc_num).amount,height);
col = 1:width;
newFT(:,col) = interp1(x,oldFT(:,col),xi,'spline',0);
handles.data.FT{frame} = rev_fftshift(newFT);
```

Listing 5.22: Rhythmic Frequency Range Compression/Expansion in `process_spec_stretch`

5.8.4 Fixed Rhythmic Frequency Stretching Processes

As with pitch shifting, there are two fixed processes that perform rhythmic frequency range adjustments without using interpolation. These processes are referred to in the software

tool as 'Halve Rhythm' and 'Double Rhythm' respectively and are defined in the function `process_fixed_rhythm` which uses the Boolean argument `up` to determine whether the rhythmic frequency range should be stretched or compressed by a factor of 2.

The operations performed on the 2D Fourier data of each frame are the same as in listing 5.21 except that the adjustment is made in the vertical dimension, on the columns, instead of the horizontal dimension, on the rows.

5.9 Resizing the 2D Fourier Spectrum

This process adjusts the dimensions of the 2D Fourier spectrum to transform the audio signal. When one of the spectrum dimensions is resized, the centre frequencies of each bin on that axis is changed. The data is interpolated to the new array size so that the signal components are still at the correct frequency.

When there is more than one frame of 2D spectrum data, resizing cannot be performed since it would result in different frame sizes for each frame and in the current software tool design this is not possible. Therefore resizing the spectrum is limited to rhythmic analysis mode, where it is guaranteed that there will only be 2D spectrum. It is recommended that in future work (chapter 9) this operation is used to transform signals that have been analysed in terms of their pitch, since it seems there is a lot of potential in this technique.

The spectrum resizing process, and the fixed resize operations (section 5.9.5) are defined last in the `proc_names array`, so that if they are not present in the context menu, no out of bounds errors are caused by indexing the context menu's `proc` array for other processes.

In rhythmic mode, it was thought that adjusting the width of the spectrum would change the duration of each row of raster image data whilst maintaining the original pitch. This would essentially alter the tempo of the signal. It was predicted that adjusting the height of the spectrum would change the duration of the signal whilst maintaining the same tempo because the same rhythmic frequency components would be oscillating over a larger number of raster image rows. It adjusts the number of beats/bars of the signal leaving the tempo and rhythm unchanged. The resize process therefore attempts to allow independent adjustment of the tempo and the duration in terms of rhythmic beats (the rhythmic duration).

5.9.1 Resize Process Parameters

The resize process has several parameters that need to be initialised in the `create_spec_resize` function, when the process is initialised. Most of them are required to display to the user, in the resize process' GUI window rather than to run the process. The process variables are described in table 5.4.

Variable	Data Type	Default Value	Description
`type`	String	'Resize Spec.'	The name of the process, obtained from the `proc_names` array. Used to identify the process as a resize operation.
`bypass`	Boolean	`false`	Determines whether or not the filter process should be bypassed when the chain of transformations is run.
`min_quadrant_size`	Integer array	[3 51]	This is a read-only parameter that defines the minimum allow size of a 2D spectrum quadrant.
`size`	Integer array	n/a	The output size of the 2D spectrum quadrant according to the current settings. Initialised as the input size.
`orig_size`	Integer array	n/a	The input size of the 2D spectrum quadrant, which is read-only.
`change`	Integer array	[0 0]	The required change in 2D spectrum quadrant size.
`tempo`	Double	n/a	The tempo of the output signal, initially set to the value defined in the `analysis_settings` structure.
`ndivs`	Integer	n/a	The duration of the output signal in terms of tempo divisions. It is initially set to the height of the raster image multiplied by the numerator of the tempo division.

Table 5.4: Parameters of Spectrum Resizing Process Set in `create_spec_resize`

The variable `ndivs` is the rhythmic duration of the signal, given in terms of the note duration or tempo division used to define the raster image width in analysis. As an example, for an audio signal that is 4 bars long, with a raster image width of a quarter-note, `ndivs` would be 16. This variable is initially set to the height of the Fourier data array, `FT`, minus the height padding, `height_pad`, (i.e. the raster image height) multiplied by the numerator of the raster width tempo division, `tempo_div_num`.

The spectrum size is considered in terms of a single quadrant, including the DC row and column, rather than the total spectrum size. This ensures that the positive and negative frequencies remain symmetrical for both frequency axes. The input spectrum size can easily be calculated from the `FT` data array but since it is required frequently in the adjust function, it is stored in the process data structure. The values of the `min_quadrant_size` array were chosen to yield a minimum signal duration of just under 100 ms, since this is nearing the threshold of human auditory perception [27].

5.9.2 Adjusting Resize Process Parameters

The GUI window for the resize process, shown in figure 5.10, allows the user to specify the required quadrant size of the output spectrum. Alternatively the user can enter the required tempo or rhythmic duration and the nearest quadrant size to this value will be set.

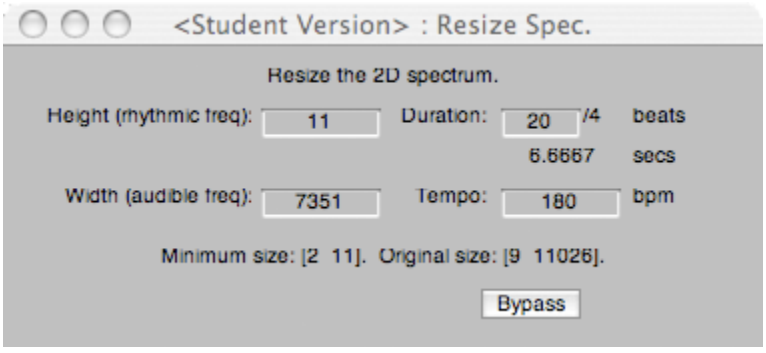


Figure 5.10: Resize Spectrum GUI

The function `adjust_spec_resize` creates the resize spectrum GUI and defines the callback functionality of the GUI objects, allowing the properties of the resize spectrum process to be

set. The 'Minimum size' values displayed on the GUI are given by the `min_quadrant_size` variable and the 'Original size' values are given by `orig_size`. 'Height' and 'Width' display the required quadrant dimensions of the output spectrum, as stored in `size`.

The signal duration is displayed in seconds, given by the `dur` variable in the `audio_settings` structure, and also in terms of rhythmic duration. The numerator of the rhythmic duration is given by the `ndivs` variable and the denominator is set to the value stored in `tempo_div_den` variable within the `analysis_settings` structure. The tempo value displayed is the tempo of the signal produced from the output spectrum, as given by the `tempo` variable.

The function `height_changed` is called when either the height value or the rhythmic value is changed. The callback for the height value's text edit object rounds the numerical input to a positive integer and stores it in `size(1)` before calling `height_changed`. The rhythmic duration value's callback function stores the absolute value of the numerical input in `ndivs` and then calculates the quadrant height that represents the rhythmic duration closest to this value, using the following equation before calling `height_changed`:

$$\texttt{size(1)} = \texttt{ceil}\left(\frac{\texttt{ndivs} + \texttt{height\_pad}}{2}\right) \tag{5.11}$$

The `height_changed` function is displayed in listing 5.23. It first ensures that the height settings is not below the minimum value given by `min_quadrant_size(1)` and then sets the string of the height value's text edit to the correct height setting. The actual rhythmic duration given by the new height setting is then calculated and displayed.

```
function height_changed(handles)
    if handles.cur_process.size(1)<handles.cur_process.min_quadrant_size(1)
        handles.cur_process.size(1) = ...
            handles.cur_process.min_quadrant_size(1);
    end
    set(handles.resize_popup.edit_height,'String',...
        num2str(handles.cur_process.size(1)));
    % calculate ndivs
    handles.cur_process.ndivs = handles.cur_process.size(1)*2 - 1  - ...
        handles.data.spec2D_settings.height_pad;
    set(handles.resize_popup.edit_duration_beats,'String',...
```

```
        num2str(handles.cur_process.ndivs));
    changed(handles);
end
```

Listing 5.23: Dealing With A Change In Height Setting Using the `height_changed` Function

The function `width_changed` is called when the width or tempo values are adjusted. The callback function for the width text edit object gets the numerical value entered and rounds it to a positive integer to store in `size(2)` before calling `width_changed`. The callback for the tempo text edit object gets the given numerical value and stores it in the process' `tempo` variable before calculating the spectrum quadrant width that most closely approximates this tempo. This width is obtained by the following equations before calling `width_changed`:

$$\texttt{im\_width} = \texttt{round}\left(\frac{240 * \texttt{tempo\_div\_num} * \texttt{Fs}}{\texttt{tempo} * \texttt{tempo\_div\_den}}\right) \tag{5.12}$$

$$\texttt{size(2)} = \texttt{ceil}\left(\frac{\texttt{spec\_width} + \texttt{im\_pad}}{2}\right) \tag{5.13}$$

The width of the output raster image is calculated first using the tempo division, the sample rate, and the current tempo value, then the output spectrum quadrant width is determined from this. The `width_changed` function performs a similar sequence of operations to `height_changed`. It checks that the current width setting is above the `min_quadrant_size(2)` value and if not corrects it, then the width display is updated in the text edit object. The actual tempo is then recalculated according to the width setting in `size(2)`, using the equation:

$$\texttt{tempo} = \frac{240 * \texttt{tempo\_div\_num} * \texttt{Fs}}{(2 * \texttt{size(2)} - 1 - \texttt{width\_pad}) * \texttt{tempo\_div\_den}} \tag{5.14}$$

Where $2 * \texttt{size(2)} - 1$ defines the width of the whole 2D spectrum. The tempo value can then be displayed in the correct text edit object on the GUI.

When any parameter is changed on the GUI, the `changed` function is called, since it is called at the end of both `width_change` and`height_changed`. This function calculates the new value of the process' `dur` variable and displays it on the GUI. It also calculates the value of the `change` array as $\texttt{size} - \texttt{orig\_size}$. Finally the `process_changed` variable in the `processes` structure is set to true and the `handles` structure is set as the resize process figure's GUI data.

5.9.3 Running the Resize Spectrum Process

The function `process_spec_resize` is used to resize the spectrum according to the resize process parameters, as stored in the `process` array. The function contains three components, resizing in the rhythmic dimension, resizing in the audible dimension and updating the analysis settings according to the new spectrum size. The function resizes the two dimensions of the spectrum in the order that will result in the least number of calculations, using the instructions shown in listing 5.24.

```
if handles.processes.process(process_num).change(2) < 0
    handles = do_audible_dim(handles,process_num);
end
if handles.processes.process(process_num).change(1) ~= 0
    handles = do_rhythmic_dim(handles,process_num);
end
if handles.processes.process(process_num).change(2) > 0
    handles = do_audible_dim(handles,process_num);
end
```

Listing 5.24: Determining The Order Of Dimension Resizing in `process_spec_resize`

The width of the spectrum is generally larger than the height, so if the spectrum width is to be decreased then this is done first using `do_audible_dim`. This means that fewer columns need to be resized in the `do_rhythmic_dim` function. If the width is to be increased then it is done after the height. The functions `do_rhythmic_dim` and `do_audible_dim` perform similar operations, though for different frequency axes. Only the resizing of the rhythmic frequency axis is discussed here to prevent repetition.

The `do_rhythmic_dim` function is displayed in listing 5.25. It uses Matlab's `interp1` function to resize the complex Fourier spectrum data using cubic spline interpolation. The width and original height of the spectrum are obtained from the `FT` data array dimensions, and the new height is calculated from the process variable `size(1)`. The new Fourier data array can then be initialised with zeros, and the original data quadrant-shifted using `fftshift`.

The arrays `x` and `xi` represent the sample point indices of the original and interpolated Fourier data columns respectively. The `xi` array has the same range as `x` but a different

length, meaning that `interp1` stretches/compresses each data column to fit the new size.

```
function handles = do_rhythmic_dim(handles,process_num)
    width = size(handles.data.FT{1},2);
    old_height = size(handles.data.FT{1},1);
    new_height = 2*handles.processes.process(process_num).size(1) -1;
    newFT = zeros(new_height,width);
    oldFT = fftshift(handles.data.FT{1});
    x = linspace(1,old_height,old_height);
    xi = linspace(1,old_height,new_height);
    col = 1:width;
    y = oldFT(:,col);
    newFT(:,col) = interp1(x,y,xi,'spline');
    handles.data.FT{1} = rev_fftshift(newFT);
    if handles.data.analysis_settings.sync
        old_im_height = old_height-handles.data.spec2D_settings.height_pad;
        new_im_height = new_height-handles.data.spec2D_settings.height_pad;
        x = linspace(1,old_im_height,old_im_height);
        xi = linspace(1,old_im_height,new_im_height);
        handles.data.analysis.period = ...
            interp1(x,handles.data.analysis.period,xi,'spline');
        handles.data.analysis.num_periods = ...
            interp1(x,handles.data.analysis.num_periods,xi,'spline');
    end
end
```

Listing 5.25: Resizing The Rhythmic Frequency Dimension

If the signal data has been analysed in pitch-synchronous mode, then pitch related analysis settings must be adjusted after the new `FT` array has been obtained. The `period` and `num_periods` arrays are also interpolated to the correct size, allowing the transformed data to be resynthesised and converted back to a 1D audio representation.

Once both frequency axes have been resized in this way, the signal analysis parameters need to be updated to reflect the new spectrum size, as shown in listing 5.26.

```
    handles.data.analysis.frame_size = size(handles.data.FT{1},2)-handles.
        data.spec2D_settings.width_pad;
    handles.data.analysis.imwidth = handles.data.analysis.frame_size;
```

```
handles.data.analysis.num_frames = size(handles.data.FT{1},1)-handles.
    data.spec2D_settings.height_pad;
handles.data.analysis_settings.frame_size_secs = handles.data.analysis.
    frame_size/handles.data.audio_settings.Fs;
handles.data.analysis_settings.tempo = (240*handles.data.
    analysis_settings.tempo_div_num)...
    /...
    (handles.data.analysis_settings.frame_size_secs*handles.data.
        analysis_settings.tempo_div_den);
handles.data.audio_settings.dur = handles.data.analysis.num_frames*
    handles.data.analysis_settings.frame_size_secs;
```

Listing 5.26: Adjusting Analysis Parameters After Resizing The Spectrum in `process_spec_resize`

5.9.4 Recalculating Resize Process Properties

Some parameters of the resize spectrum process are related to the size of the unprocessed 2D Fourier data array. When the signal analysis is changed (section 4.6.4), these parameters need to be updated accordingly.

The function `calc_resize_properties` is used to adjust the parameters of a resize process after analysis settings are changed. If the analysis mode is changed from rhythmic to timbral, then the resize process must be removed from the process chain entirely, using the same operations described in section 5.1.3. If the signal representation is still in rhythmic mode then the size of the new input 2D Fourier spectrum array must be determined and stored in `orig_size`. The output 2D spectrum dimensions (`size`) can then be determined using the `change` array and `orig_size` according to the equation:

$$\texttt{size} = \texttt{orig\_size} + \texttt{change} \tag{5.15}$$

Finally the new values in `size` are checked against `min_quadrant_size` to ensure that the spectrum is not smaller than the minimum allowed spectrum size.

5.9.5 Fixed Spectrum Resize Processes

There are four fixed processes available in the software tool that allow resizing of the spectrum: 'Double Dur.', 'Halve Dur.', 'Double Tempo' and 'Halve Tempo'. The operation of these processes is defined by the functions `process_dur` and `process_tempo`, which both have the Boolean input argument `double` to determine whether to perform the respective 'Double' or 'Halve' operation. These processes were written to investigate the effects of tempo and rhythmic duration adjustments without the influence of interpolation error. Doubling and halving of the 2D spectrum dimensions can be performed simply by inserting zero values between each point and removing every other point. Listing 5.27 shows the array operations that define the new Fourier data for the 'Double Dur.' and 'Halve Dur.' processes within the `process_dur` function.

```
        if double
            newFT = zeros((height*2 - 1),width);
            row = 1:height;
            newFT((2*row -1),:) = oldFT(row,:);
        else
            newFT = zeros(ceil(height/2),width);
            row = 1:ceil(height/2);
            newFT(row,:) = oldFT((2*row - 1),:);
        end
        handles.data.FT{1} = rev_fftshift(newFT);
```

Listing 5.27: Doubling/Halving Rhythmic Duration in `process_dur_resize`

The 'Double Tempo' and 'Halve Tempo' processes are performed using the same operations on the width, rather than the height. However 'Double Tempo' requires halving of the spectrum width and 'Halve Tempo' requires the width to be doubled, as shown in listing 5.28.

```
        if double
            newFT = zeros(height,ceil(width/2));
          col = 1:ceil(width/2);
            newFT(:,col) = oldFT(:,(2*col - 1));
        else
```

```
            newFT = zeros(height,(width*2 - 1));
        col = 1:width;
            newFT(:,(2*col - 1)) = oldFT(:,col);
    end
    handles.data.FT{1} = rev_fftshift(newFT);
```

Listing 5.28: Doubling/Halving Tempo in `process_tempo_resize`

After the new Fourier data array has been calculated the signal analysis parameters must be updated appropriately as in the `process_spec_resize` function.

5.10 Inversion of the 2D Fourier Spectrum

This is a simple fixed process that uses the `fftshift` function to invert the frequency content of the 2D Fourier spectrum so that components near the centre of the spectrum, with low rhythmic and audible frequency, are shifted towards the corners of the spectrum, and components at the corners of the spectrum, with high rhythmic and audible frequency, are shifted towards the centre of the spectrum. It was thought that the transformation might provide an interesting perceptual effect.

Chapter 6

Evaluation Two-Dimensional Audio Analysis and Processing in Matlab

Now that the 2D Fourier analysis and processing capabilities of the software tool have been fully desribed, it is important to evaluate these techniques and identify the conditions in which they are most appropriately used. A variety of audio signals have been analysed in the software tool, investigating their properties using 2D representations in both time and frequency domains and revealing the capabilities and limitations of the developed techniques. The 2D Fourier domain transformation processes have been explored using different audio signals. Their operation can now be analysed and appropriate applications can be identified to achieve interesting transformations.

6.1 Effects of Raster Image Width on 2D Fourier Analysis

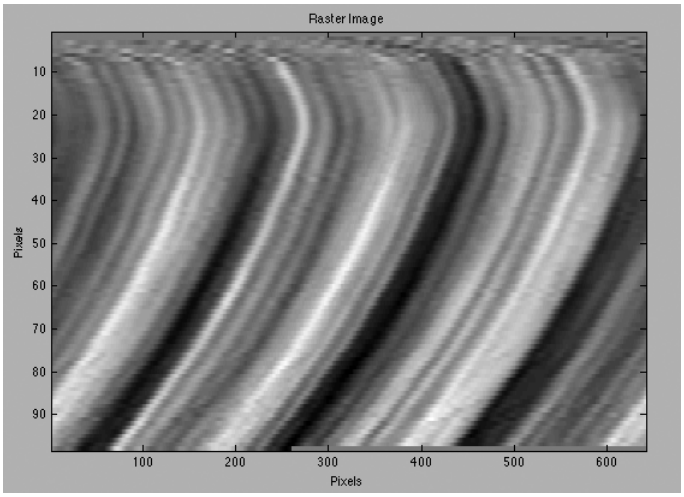
The importance of applying the correct raster image width was explained in section 4.3.2. If the centre frequency points of the 2D Fourier analysis bins are correctly aligned with frequency components of the signal data then these components can be defined by a single point in the spectrum yielding a clearer data representation. Depending on the analysis mode, the raster image dimensions are either determined by the pitch or tempo of the audio signal. This section demonstrates the benefits of correctly defining the relevant analysis

parameter.

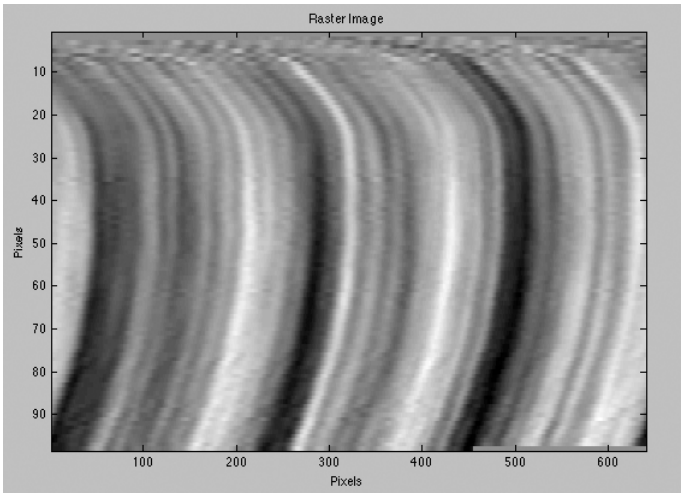
6.1.1 Determining Correct Pitch

When the raster image width is defined according to the fundamental frequency period, signals often exhibit patterns that depict slow pitch variations. The analysed pitch is correct when these patterns align vertically within the raster image. This is demonstrated for a cello playing a C# in figure 6.1.

The pitch was calculated at 68.6916 Hz giving an image width of 642 pixels, however figure 6.1a shows that this doesn't properly represent the pitch of the signal, since the periodic signal pattern is displayed at an angle. The image width was adjusted to 639 pixels using the 'Info' GUI window, this corresponds to a pitch of 69.0141 Hz. Figure 6.1b shows how this corrected pitch setting aligns the periodic signal vertically in the raster image, apart from at the start and end of the signal where the pitch increases and decreases respectively.



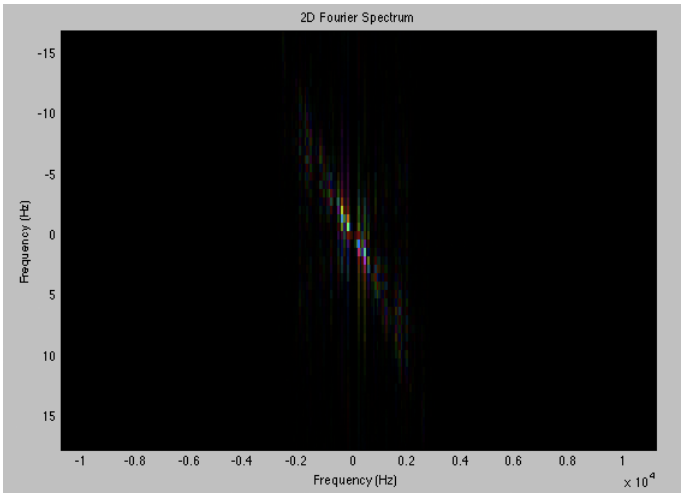
(a) Calculated Pitch: 68.6916 Hz - Raster Image Width: 642 samples



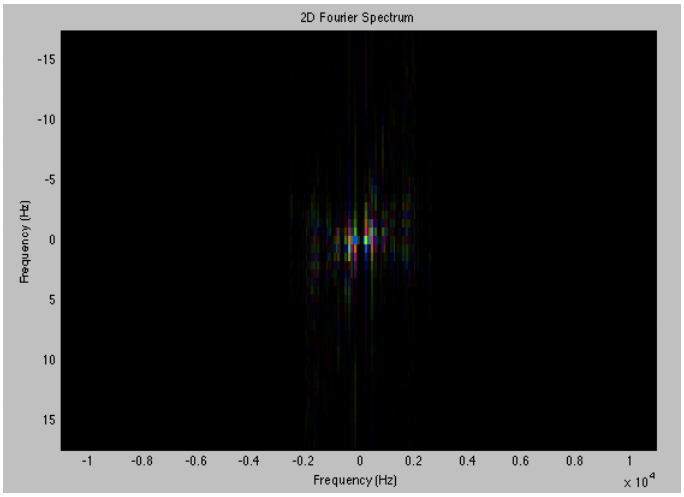
(b) Corrected Pitch: 69.0141 Hz - Raster Image Width: 639 samples

Figure 6.1: Correcting Pitch Analysis For A Cello Playing C#

Correcting the pitch analysis settings to obtain vertically aligned signal periodicities will ensure a more concise 2D Fourier spectrum since the signal components will be aligned with the centre frequencies of analysis bins, preventing the spectral energy from spreading across several bins. This is demonstrated by the spectrum of the cello sound when the pitch analysis is corrected, as shown in figure 6.2.



(a) Calculated Pitch: 68.6916 Hz - Raster Image Width: 642 samples



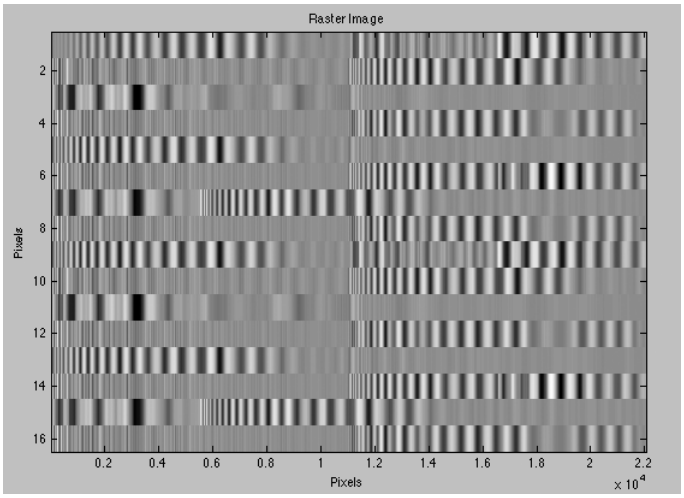
(b) Corrected Pitch: 69.0141 Hz - Raster Image Width: 639 samples

Figure 6.2: Improving 2D Spectrum By Correcting Pitch Analysis

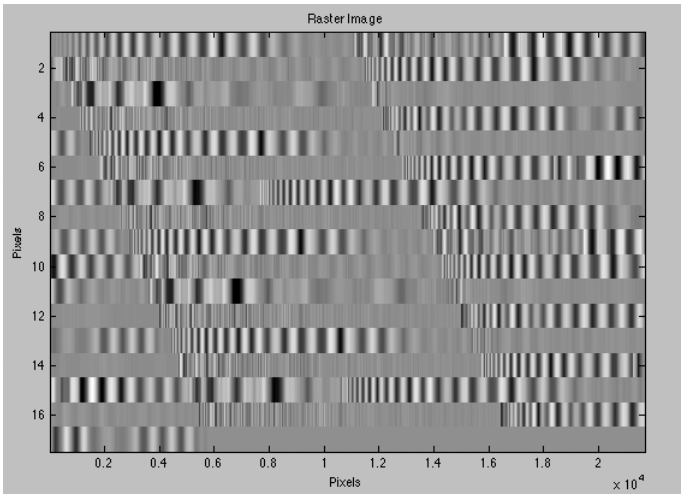
6.1.2 Determining Correct Tempo

The scale of raster image dimensions is quite different in rhythmic analysis mode, where the width is defined according to a tempo-based duration. However the same principal applies as with timbral analysis. When the tempo of the audio signal is correctly defined, the signal waveform will demonstrate vertically aligned patterns in the raster image. This

is demonstrated with an electronic drum beat in figure 6.3.



(a) Correct Tempo: 120 bpm - Raster Image Width: 22050 samples



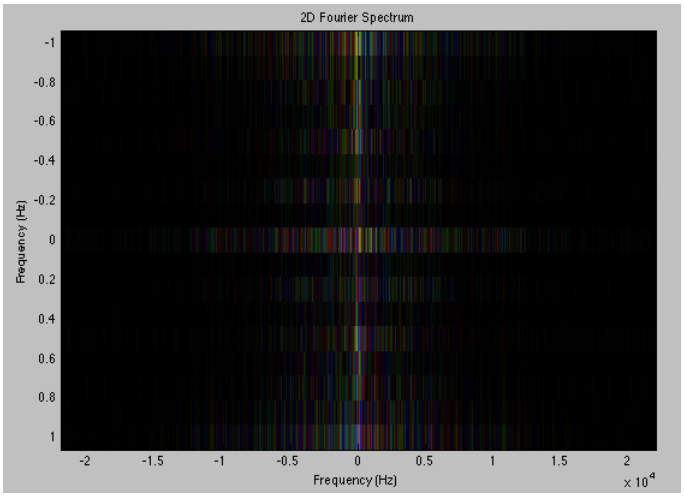
(b) Incorrect Tempo: 122 bpm - Raster Image Width: 21689 samples

Figure 6.3: Determining Correct Tempo In The Raster Image

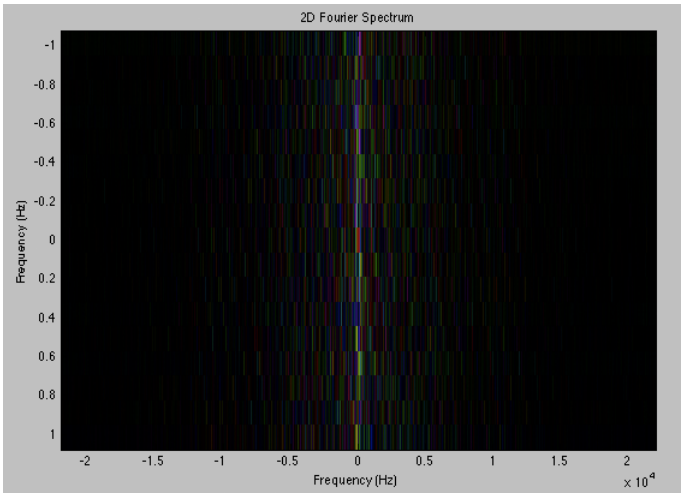
The `mirtempo` function correctly calculated the tempo of the audio signal as 120 bpm. One indication that this is the correct value is the tempo based duration in the analysis settings GUI. The loop is known to contain exactly four bars so when the duration is displayed as 16/4 beats, the tempo must be set correctly. Figure 6.3a shows the raster image for this audio signal, loop120.wav, at the correct tempo with a row size set to 1/4

beats (a quarter-note). The image clearly displays vertically aligned periodicity due to the repetitive elements of the drum rhythm. When the tempo is adjusted to 122 bpm in figure 6.3b, the drum hits are no longer vertically aligned but skewed and as a result the raster image gives a much less intelligible representation of the signal.

When the tempo is correctly set, the 2D Fourier spectrum representation is clear (figure 6.4a). Some rows of the spectrum display clearly display many points of large magnitude. These rows correspond to rhythmic frequencies that are prominent within the drum rhythm. Other rows have much lower average magnitudes, since they do not represent periodic rhythmic components in the signal.



(a) Correct Tempo: 120 bpm - Raster Image Width: 22050 samples



(b) Incorrect Tempo: 122 bpm - Raster Image Width: 21689 samples

Figure 6.4: Effect Of Correct Tempo Analysis On 2D Spectrum

The 2D Fourier spectrum of the drum loop is much less intelligible when the tempo is not correctly set. The energy of 2D spectral components cannot be precisely represented by frequency points in either dimension and so their magnitude energy is spread between adjacent points. This spreading of energy is shown by the spectrum display in figure 6.4b.

6.2 Rhythmic Mode Analysis

Rhythmic analysis mode provides a clear display of the sub-sonic signal variations in both the time and frequency 2D representations. For audio signals with simple and repeating rhythms, the rhythmic frequencies of 2D Fourier spectrum components are well-defined.

A simple drum kit rhythm was programmed using a MIDI sequencer, as displayed in figure 6.5. This MIDI pattern was used to create the audio file simple120.wav at a tempo of 120 bpm and for a duration of 4 bars with Native Instruments Battery 3's basic drum kit samples. The rhythm repeats every 2 crotchet beats i.e. a rhythmic duration of 1/2 and all MIDI velocities were set to the same value to remove any subtle rhythmic variation from the signal.

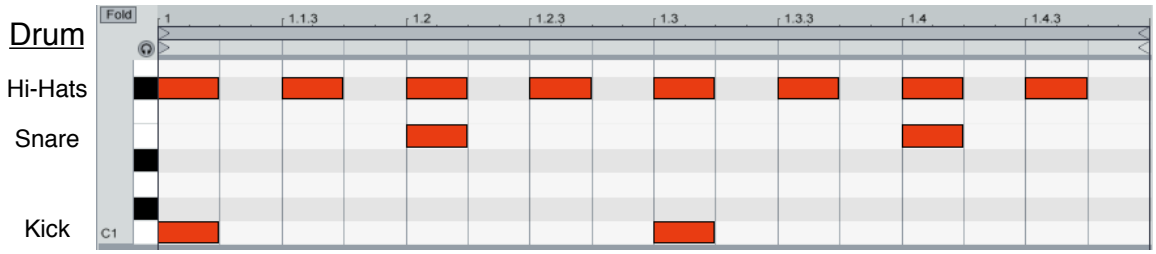
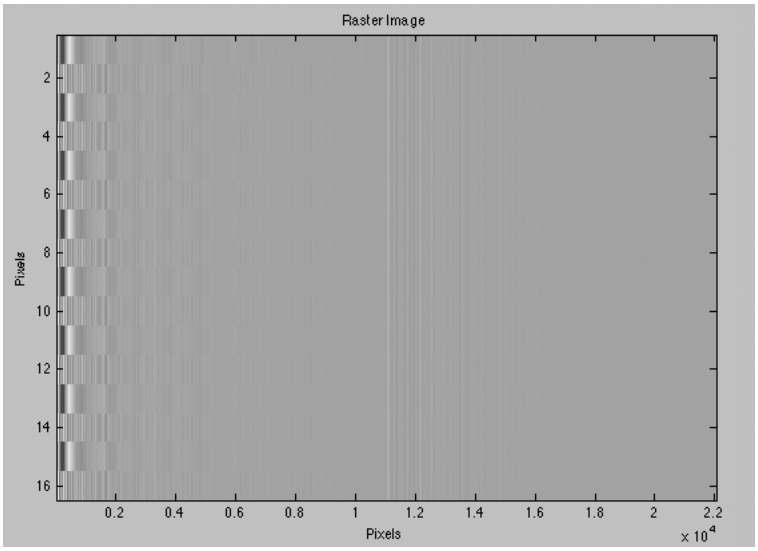
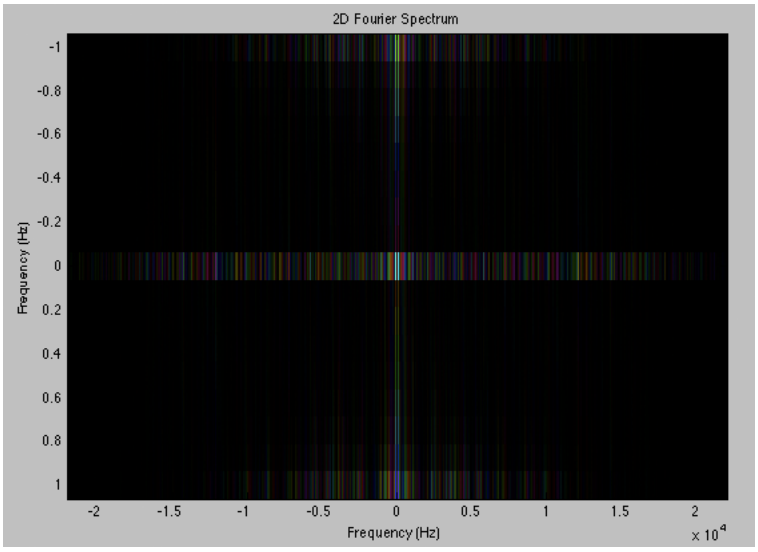


Figure 6.5: A Simple Drum Loop Programmed in a MIDI Sequencer

This audio signal was then imported into the software tool and analysed at the known tempo with a row size of 1/4 beat. The resulting 2D representations of simple120.wav are shown in figure 6.5.



(a) Raster Image



(b) 2D Fourier Spectrum

Figure 6.6: 2D Analysis of simple120.wav With 1/4 Beat Row Size

The 2D spectrum display clearly shows that most rhythmic frequency energy contained in the rows with rhythmic frequency centred at 0 Hz and 1 Hz. The 0 Hz row represents the audible frequency components that are the same in every row of the raster image. The 1 Hz row corresponds to a rhythmic variation with a period of one half-note. Notice how the spectral components with 0.5 Hz and 0.25 Hz rhythmic frequency (1 bar and 2 bar periods at 120 bpm) have little energy in this spectrum, but the spectrum of the

audio signal loop120.wav contains a lot of energy in this row (figure 6.4a). This is because simple120.wav is exactly repeated every half-note whereas loop120.wav has varies subtly over the four bars.

6.2.1 Changing Rhythmic Frequency Range

When row size is altered according to rhythmic duration, it changes the rhythmic frequencies that are represented by the points of the 2D spectrum's vertical axis. The maximum frequency point on the rhythmic axis corresponds to a rhythmic periodicity that is double that of the row size. In other terms, the maximum frequency of the rhythmic axis is half the rhythmic sampling frequency.

Rhythmic sampling frequency is given by the equation:

$$f_{rs} = f_s/N \tag{6.1}$$

The interval between rhythmic frequency points is given by:

$$\Delta f_r = f_s/MN \tag{6.2}$$

Where M and N are the width of the 2D spectrum.

When the duration of the raster image width is halved, for example from a quarter-note to an eighth-note, the rhythmic sampling frequency is therefore doubled. To obtain 2D spectral components with higher rhythmic frequency, the row width must be reduced, as demonstrated in figure 6.7 for simple120.wav with a row width of an eighth-note.

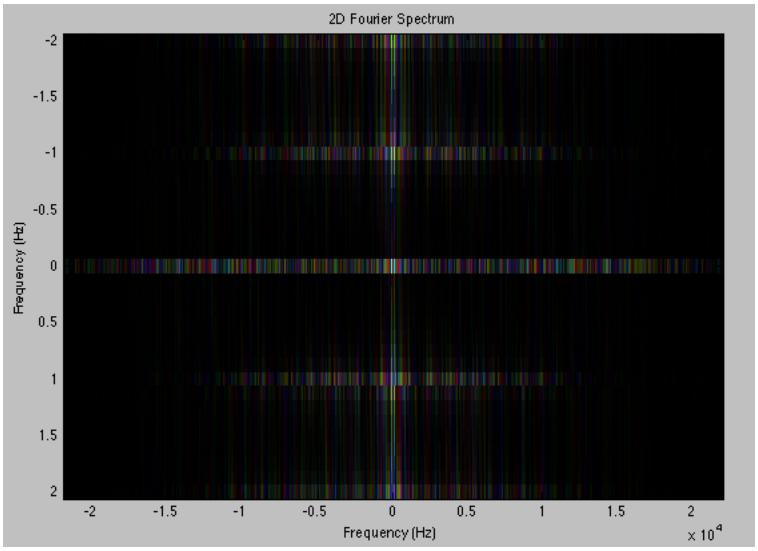


Figure 6.7: 2D Analysis of simple120.wav With 1/8 Beat Row Size

This spectrum has a rhythmic frequency ranging up to 2 Hz, which is a quarter-note rhythm at 120 bpm. There are many 2D spectral signal components of 2 Hz, since the kick and snare drum hits occur alternately at this frequency. The rhythmic frequency envelope in the range of 0 Hz to 1 Hz is the same as in figure 6.6 though with half as many points. The signal's rhythmic frequencies are less well defined between 1 Hz and 2 Hz since none of the analysis frequencies correspond precisely to rhythmic variations in the signal.

6.2.2 Limitations of Integer Raster Image Width

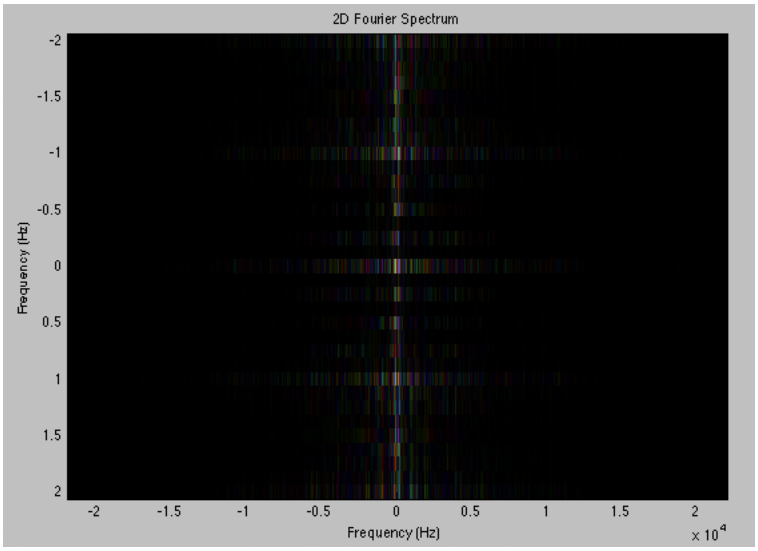
After experimenting with audio signals in rhythmic analysis mode it is apparent that the data sample rate limits rhythmic synchronisation in the 2D Fourier analysis. The raster image width must be an integer since no resampling functionality has been provided.

A very limited set of tempo values give integer image widths, for example a quarter-note duration at 121 bpm corresponds to a period of 21867.768 samples (to 3 decimal places). Even tempos that do allow integer image widths, reach their limit as the rhythmic duration of the image width is decreased. At 120 bpm, integer widths cannot be specified at a duration one sixteenth-note or less.

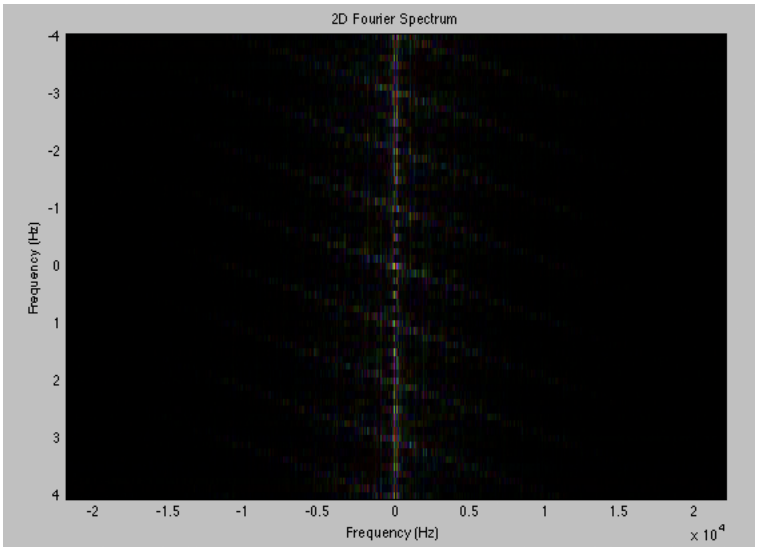
Within the software tool, when the required raster image width is not an integer, the

nearest integer value is used. The analysis is therefore no longer perfectly rhythmically-synchronised and the centre frequencies of the rhythmic axis points no longer match sub-sonic sinusoidal variations of the signal. Hence the clarity of the 2D Fourier spectrum analysis is reduced

This is issue is demonstrated in figure 6.8. Figure 6.8a shows the 2D spectrum of the audio signal loop120.wav for a row width of an eighth-note, which can be represented by an integer. The spectrum in figure 6.8b represents the same signal but with a required row width of a sixteenth-note. This rhythmic duration has a period of 5512.5 ($f_s/8$) which has to be rounded to 5513. The spectrum shape is therefore skewed and smeared since the rhythmic frequency bins no longer precisely match the rhythmic frequencies of the 2D spectrum components.



(a) Eighth-Note Width Duration



(b) 2D Fourier Spectrum

Figure 6.8: Limitations of Raster Image Width When Reducing Row Duration

6.2.3 Pitch-Synchronised Rhythmic Analysis

The pitch-synchronised rhythmic analysis mode has some fundamental problems. The idea was to resample each row to ensure that each row contained a whole number of periods of the signal. Firstly, the signal cannot be guaranteed to start at a zero-crossing

at the beginning of each row and therefore a whole number of periods may still result in a discontinuity between the beginning and end of the row. More importantly, resampling the content of each the raster image row separately removes makes the 2D Fourier spectrum display incorrect. It is not know what frequency a component represents in the 2D Fourier representation because it's sample rate is not known. Frequency domain PSOLA processing uses a variable frame size not a variable sample rate.

Rasterisation essentially applies a rectangular frame window to the audio signal to obtain each image row. Therefore if any form of transformation has been performed, signal discontinuities occur in the audio signal at points where the raster image rows join, since the transformation process will not change each row in the same way.

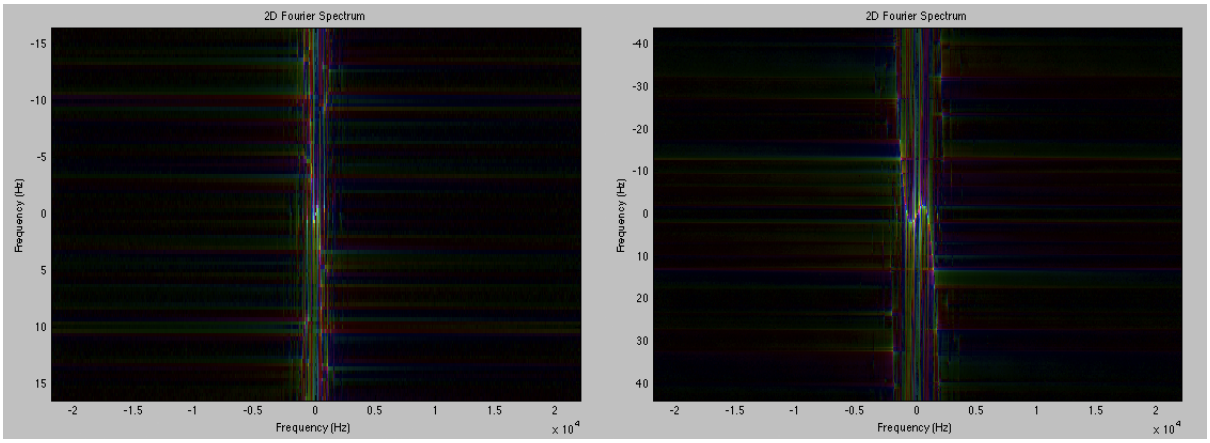
6.3 Timbral Mode Analysis

In timbral analysis mode, the 2D Fourier spectrum provides a display of the sub-sonic variation of harmonic audible frequency content of a signal. The audible frequency axis of the 2D spectrum are set to precisely match the harmonic frequencies of the signal, based on the detection of the fundamental frequency. The rhythmic frequency points do not necessarily correspond to the precise rhythmic variations within the signal and so 2D spectral components are smeared in the rhythmic axis. The rhythmic frequency interval is defined in equation 6.2 and therefore it is intrinsically related to the pitch of the signal, rather than any rhythmic analysis component.

6.3.1 Instrument Timbres

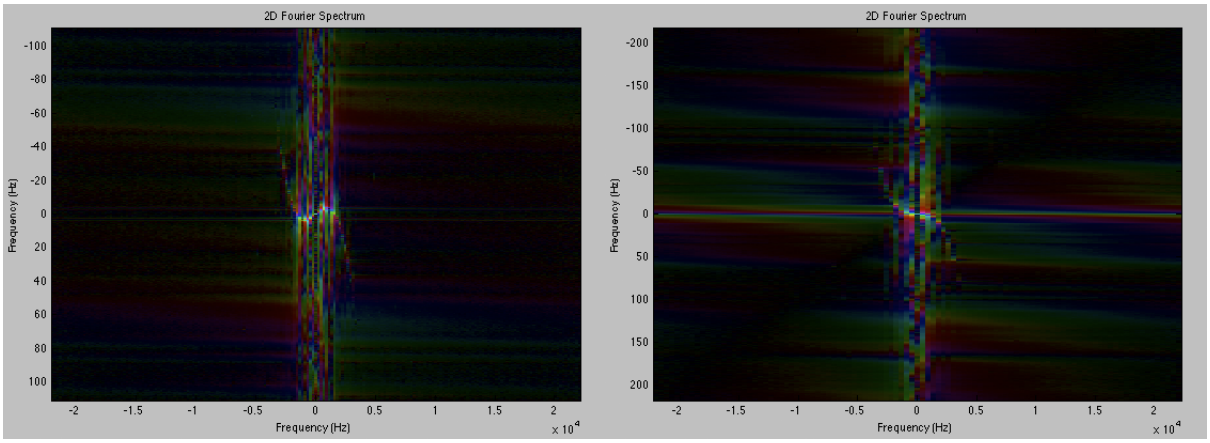
Two dimensional timbral analysis of instrument notes depicts the internal oscillations of the signal harmonics. Notes of any pitch from the same instrument will produce a similar form in the 2D Fourier spectrum. This is demonstrated in figure 6.9 which shows the 2D Fourier spectra of 3 seconds of four piano note recordings, ranging from C1 to A4. This figure shows a common pattern in all four spectra, however as the pitch increases, the harmonic content is stretched further across the spectrum image. The prominent rhythmic frequency of each harmonic follows an arc that is reflected in the diagonal of the spectrum for all of the four spectra. The 2D spectrum display's contrast was set at -50% to give a

clearer display of the spectral form.



(a) Note: C1

(b) Note: F2

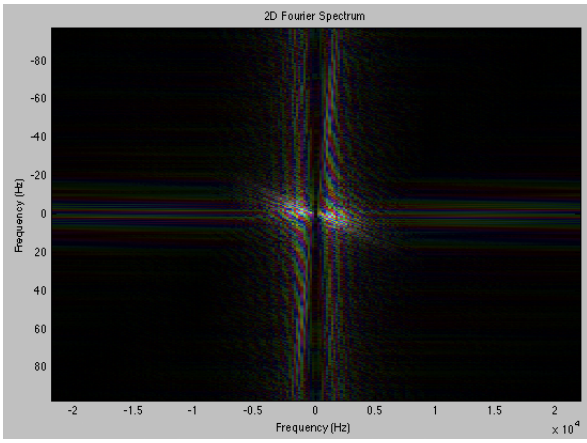


(c) Note: A3

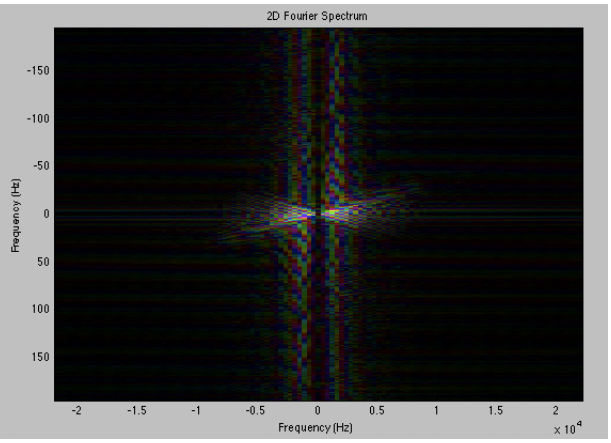
(d) Note: A4

Figure 6.9: Similar 2D Spectral Form of Piano Notes of Varying Pitch

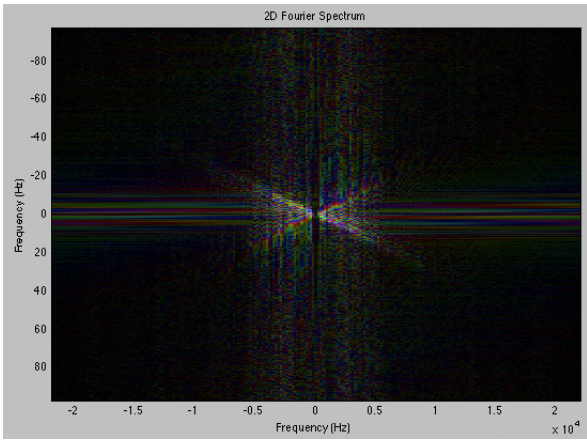
To demonstrate that other musical instruments demonstrate their own pattern of prominent rhythmic frequency of harmonic content, figure 6.10 shows the 2D Fourier spectra of two different notes from a B^b trumpet and a violin.



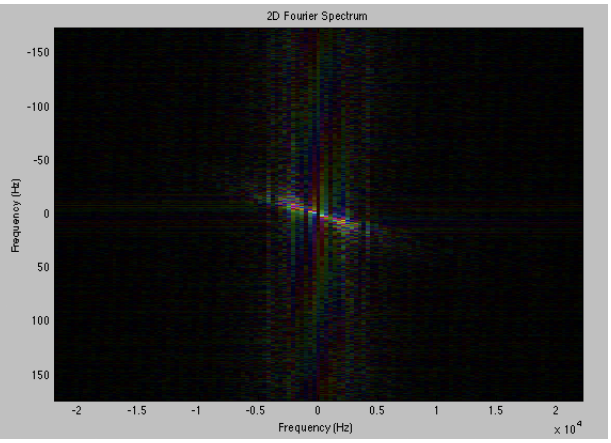
(a) B^b Trumpet Note: G3



(b) B^b Trumpet Note: G4



(c) Violin Note: G3 (Arco)



(d) Violin Note: G4 (Arco)

Figure 6.10: Similar 2D Spectral Form of Instrument Notes of Varying Pitch

6.3.2 Limitations of Integer Raster Image Width

Figure 6.10 also demonstrates the limitations of the signal sample rate in accurately synchronising the analysis to the signal pitch. The actual pitch of an audio signal will rarely correspond to an integer period size and therefore the audible frequency analysis cannot be guaranteed to synchronise the centre frequency of audible frequency analysis bins with harmonic components of a signal.

The spectra in figure 6.10 show skewing according to the angle of the waveform similarities in the raster image. Figure 6.11 shows the raster image for the violin note G4, at the

nearest integer width to its actual pitch period. This representation clearly does not represent the fundamental period of the signal in each raster image row since the waveform is not vertically aligned. The corresponding spectrum of this signal, shown in figure 6.10d, demonstrates this skewing effect when compared to the correctly pitch-synchronised violin signal of figure 6.10c.

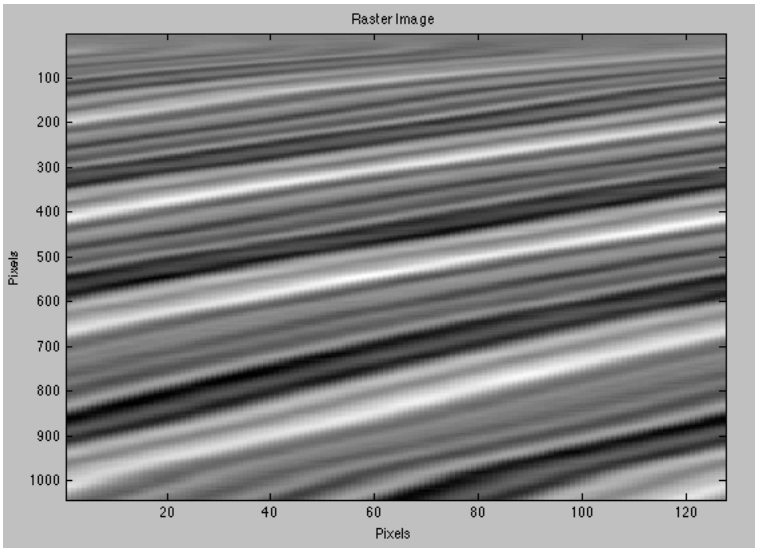


Figure 6.11: Raster Image of a Violin Note: G4 (Arco)

6.3.3 Tempo-Synced Timbral Analysis

Tempo-synced timbral analysis allows the user to specify a frame-size according to a tempo-synchronised duration. So for example, if a quarter-note duration was specified, the analysis would result in a raster image and 2D Fourier spectrum for each quarter-note of the signal at the analysis tempo. This method of analysis was designed to allow each note of a melodic signal to be analysed separately according to its pitch.

The 2D spectra of each frame will only be correctly pitch-synchronised by timbral analysis if note changes happen at the frame boundaries, since each frame has a single pitch value. The frame-size should therefore be specified according to the minimum note duration in the audio signal. The rhythmic timing of the audio signal has to be precise to allow accurate tempo-synced timbral analysis.

6.3.4 Zero Padding

The use of zero padding in the software tool's signal analysis was not well designed. In Fourier analysis zero padding is used to reduce the frequency interval between spectrum points, improving the resolution of frequency analysis.

The analysis process of this project used a single row and/or column of zero padding to ensure that the spectrum had odd dimensions (see section 4.4.4). The issue is that this resizing of the spectrum changes the centre frequency of the analysis bins, invalidating any synchronous analysis which would originally have improved the clarity of the spectral representation.

Zero padding should have been used to greatly improve the spectral resolution, applying many more rows and columns of padding, or not at all, to maintain the synchronous analysis. The drawback of using significant zero padding in a 2D signal representation is that it greatly increases memory requirements, which are already considerable larger than for a 1D representation.

6.4 2D Fourier Processing

The investigation into audio transformation using the 2D Fourier transform has provided some novel and interesting results, but there are also many limitations of the analysis data when it comes to processing. This section will describe the general features of the 2D Fourier spectrum, as obtained by the analysis techniques of this investigation, that affect the transformation of audio data. It will then give an evaluation of each of the transformation processes within the software tool. A set of audio examples are provided on the accompanying CD to this report (appendix B).

6.4.1 Analysis Limitations

Section 4.3.2 gave an analysis of the 2D Fourier transform. It stated that the 2D Fourier spectrum in not symmetrically valued in both axes since the process involves a Fourier transform of complex data. However, when properly synchronised in both axes a single amplitude-modulated sinusoidal component is represented by four symmetrically placed

points with the same rhythmic and audible frequency but different signs. Equation 4.3) shows the linearity of the DFT, which implies that linear processes can be applied to the 2D Fourier spectrum provided that each quadrant is treated in the same way. An extension of the analysis of the complex DFT (equation 4.8) could reveal the precise relationship between the positive and negative frequency data, hence allowing non-linear processing to be applied in the future.

Many of the transformation processes developed use resampling of the data to perform time/pitch modifications. After experimentation with these processes in the software tool, it is apparent that the frequency resolution of analysis can often be too low to allow accurate interpolation.

In timbral mode, the audible frequency bins are spaced at harmonic intervals of the signal when correctly pitch-synchronised. At this resolution the data will be synthesised with harmonic components at the analysis frequencies even if the data has been rescaled. There need to be many points between the harmonic frequencies to allow effective scaling of the data. However the resolution in the rhythmic frequency axis is high enough to allow scaling of frequencies by resampling. In rhythmic mode it is the rhythmic frequency interval that is at a low resolution and so resampling in this dimension is inaccurate.

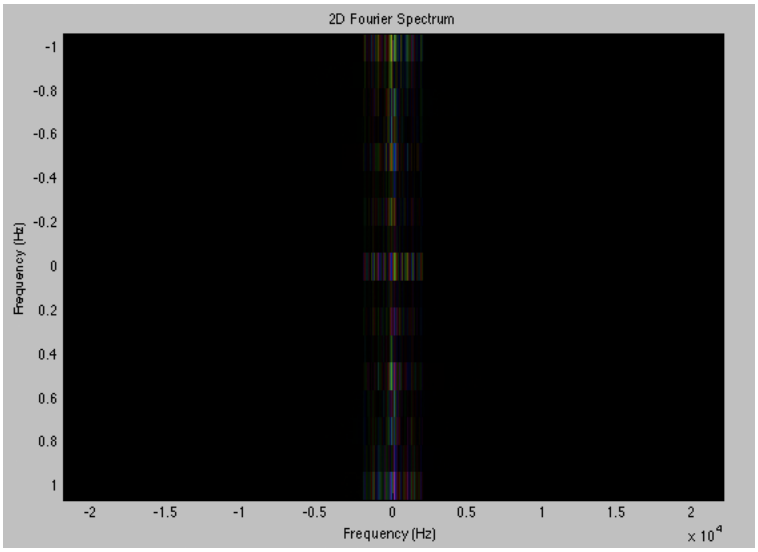
Also whilst zero padding does not affect the signal representation after the basic analysis-synthesis process, it affects the results when certain transformations are applied to the signal. Any resampling of data that reduces the period length of audible or rhythmic frequency components will introduce the added zero-valued samples into the resulting signal, since the Fourier analysis considers them to be part of the signal's period (see section 2.1.1).

6.4.2 Filtering The 2D Fourier Spectrum

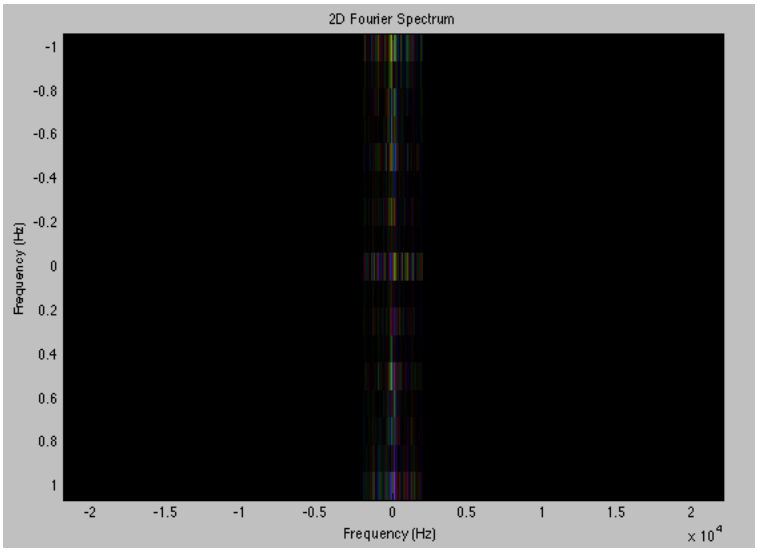
Two-dimensional spectral filtering has proven the most useful and exciting sound transformation method for creative purposes. The process is linear and operates in the same way on all four quadrants of the spectrum, therefore there are no significant analysis limitations for this process.

Filtering of audible frequencies produces very similar results to 1D frequency domain filtering. This is demonstrated in figure 6.12 where the 2D Fourier spectra are compared for

loop120.wav filtered using a 2 kHz ideal filter in both 1D and 2D frequency domains.



(a) 1D Frequency Domain Filtering loop120.wav Ideal 2kHz Low Pass Filter



(b) 2D Frequency Domain Filtering loop120.wav Ideal 2kHz Low Pass Filter

Figure 6.12: Comparison of Audible Frequency Filtering in 1D and 2D Frequency Domains

On closer inspection the signals are not identical, but this is thought to be due to a different in audible frequency resolution of the two methods, rather than the data representation itself.

In the rhythmic frequency dimension, spectral components can be filtered according to their sub-sonic variation without changing the audible frequency range, which produces unique and musically useful results.

For rhythmic analysis mode, the filtering changes the variations between rows of the raster image. When properly tempo-synchronised this alters the variation between adjacent beats of the signal. To demonstrate the process the signal loop120.wav was rhythmically filtered using a low-pass ideal filter with a cut off of 0 Hz, so that only the DC rhythmic frequency row was present. The result was an audio signal with the same audible frequency range but no variation between quarter-notes, the raster image width duration. The raster image of the resulting signal is shown in figure 6.13.

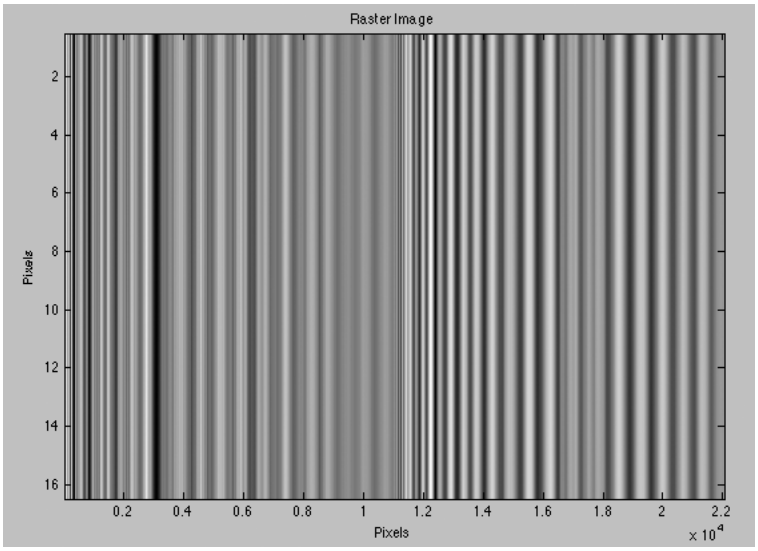
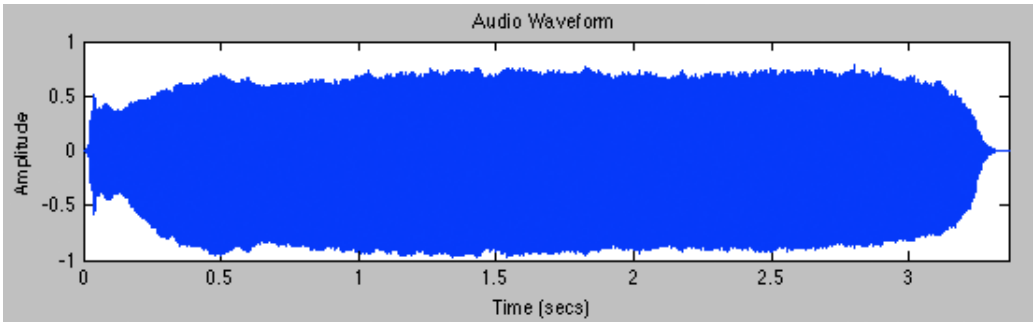


Figure 6.13: Raster Image of loop120.wav Rhythmic Frequency LP Filtered With a Cutoff of 0 Hz

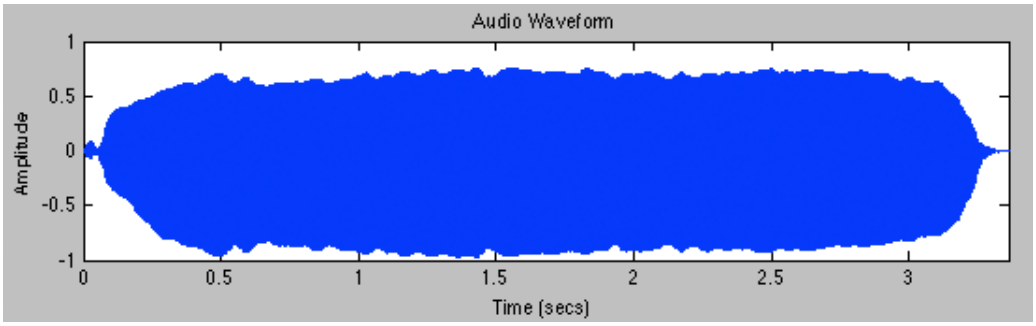
Many variations of the same rhythmic audio signal can be produced using rhythmic frequency filtering. Changing the raster image width changes the beat duration on which the filtering is based and therefore alters the output of the filtering process, even when the same filter settings are used.

In timbral mode rhythmic frequencies cover a much larger range, describing the internal sub-sonic oscillations between audible harmonics. The low frequency range corresponds to the slowly varying components that define the stable timbre of a sustained note, whilst

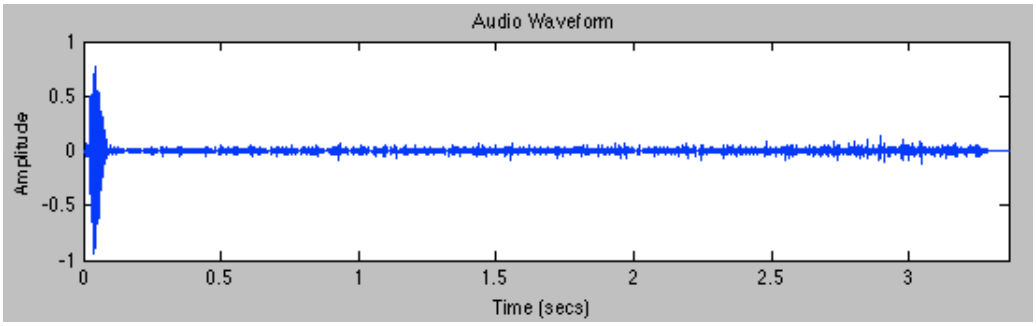
high rhythmic frequencies correspond to the note onset. The resulting audio waveforms after high-pass and low-pass rhythmic frequency filtering of a trumpet note are displayed in figure 6.14.



(a) Original Waveform



(b) 2nd Order Butterworth 15 Hz Low-Pass Filter



(c) 2nd Order Butterworth 45 Hz High-Pass Filter

Figure 6.14: Rhythmic Frequency Filtering of a B^b Trumpet Note

Low-pass filtering can therefore be used to reduce the attack of a note, below a certain point it begins to effect the oscillations defining the timbre of the note. Band-pass filtering enables interesting amplitude modulations to be observed within the signal, which could

be useful for analysis of the sub-sonic timbral variations.

Using a Butterworth frequency response has little noticable effect in the rhythmic frequency axis since the frequency resolution is generally too low. In the audible frequency axis however a Butterworth frequency response provides a more familiar and comfortable effect with less noticable resonance. Boosting of pass-band frequencies can be used to provide more subtle rhythmic variations to a signal. It's effects are less noticable in the audible frequency axis.

6.4.3 Thresholding The 2D Fourier Spectrum

Magnitude thresholding of the audible frequency columns or rhythmic frequency rows is a linear process that processes each quadrant of the 2D Fourier spectrum in the same way. This processing tool allows some interesting creative transformations of an audio signal by removing certain 2D spectral components but it can also serve as a useful analysis tool to observe the 2D spectral structure of a signal.

Rhythmic frequency thresholding allows the most dominant sub-sonic oscillations of the signal to be extracted or removed. Audible frequency thresholding allows the removal of either the strongest or weakest audible components according to the threshold settings, which in timbral analysis mode allows separation of the harmonics of the signal.

When the process applies a magnitude threshold to the spectrum according to the values of individual points, the lack of spectral symmetry is not taken into account. If the signal contains a 2D spectral component with synchronised audible and rhythmic frequency, the spectrum will contain two pairs of symmetrical points with different magnitude values. If the threshold magnitude lies between the magnitudes of the two pairs then one will be removed and the amplitude modulated sine wave will become an angled sine wave with unsynchronised pitch.

Most audio signals have more low frequency energy and so the 2D spectrum magnitude thresholding tends to separate components according to their audible frequency, unless in rhythmic frequency mode. The process could be extended by optionally scaling the threshold value across the audible frequency range to match the logarithmic pattern of human hearing.

6.4.4 Rotating The 2D Fourier Spectrum

Rotation of the 2D Fourier spectrum produces the same results as rotating the raster image would. A rotation of 90° or 270° allows the rhythmic frequency energy of the signal to be converted to audible frequency energy and vice versa. A rotation of 180° simply reverses the signal, although when the vertical axis is zero padded the zero row is not removed since it is now at the top of the image.

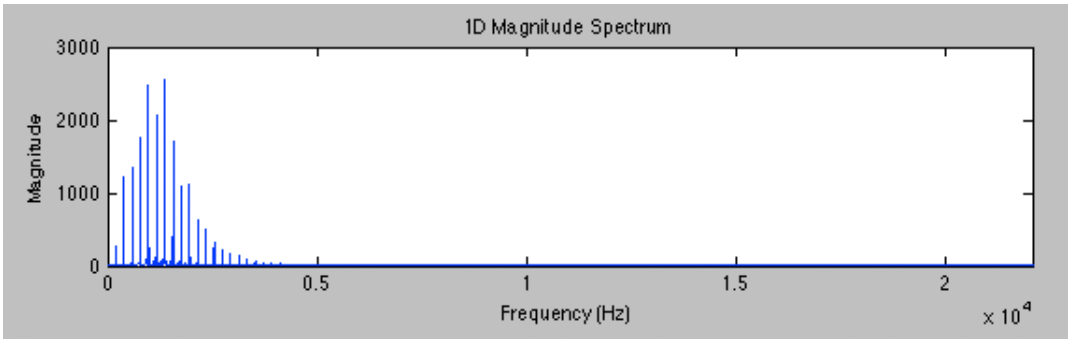
In rhythmic mode the result of 90° or 270° is composed of a few widely spaced audible sinusoids with intricate sub-sonic variations. In timbral mode the results are generally more useful because there are many more audible sinusoids, since the two spectrum dimensions are typically closer in size. The process provides somewhat interesting sounds but its musical application is limited.

Musical signals generally have much more low-frequency energy than high-frequency. To improve the results of 90° and 270° rotations, the frequencies of the audible axis could be scaled to reflect this, since there is currently an unnatural amount of high frequency energy.

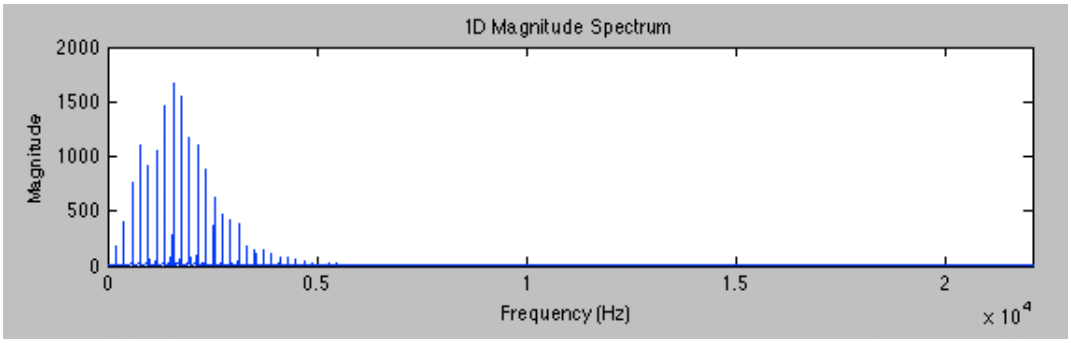
6.4.5 Pitch Shifting

The results of audio transformation using this process have been quite poor.

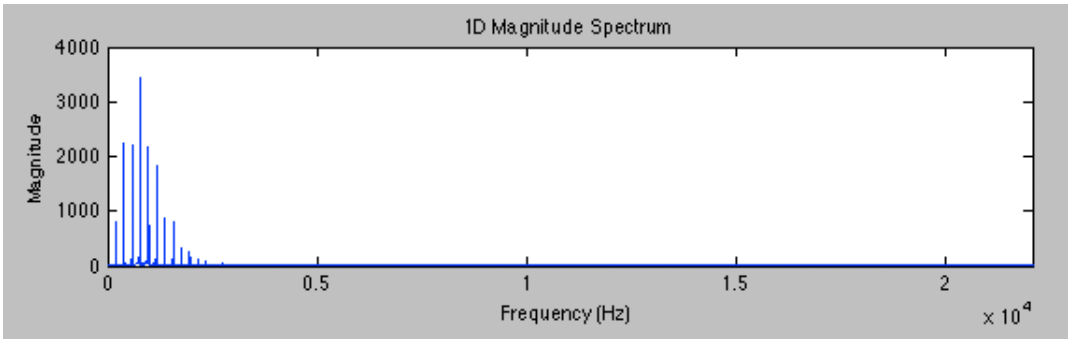
In timbral analysis mode the resolution of audible frequency analysis is too low to perform an interpolation process. Each point in the audible frequency axis is harmonically related to the fundamental frequency of the analysis which is synchronised with the pitch of the unprocessed signal. When the data has been interpolated in this axis, the resulting signal has the same pitch but the harmonic content of signal has been spread. This is demonstrated in figure 6.16, which shows the 1D Fourier magnitude spectrum for a trumpet note and the spectrum of the same signal after pitch shifting. The magnitude envelope is interpolated over a different range of values but these components are still harmonically related to the same fundamental pitch.



(a) Original Waveform



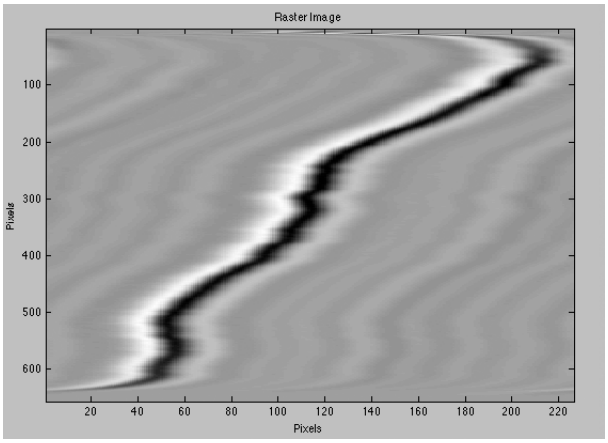
(b) Pitch Shifted Up 5 Semitones



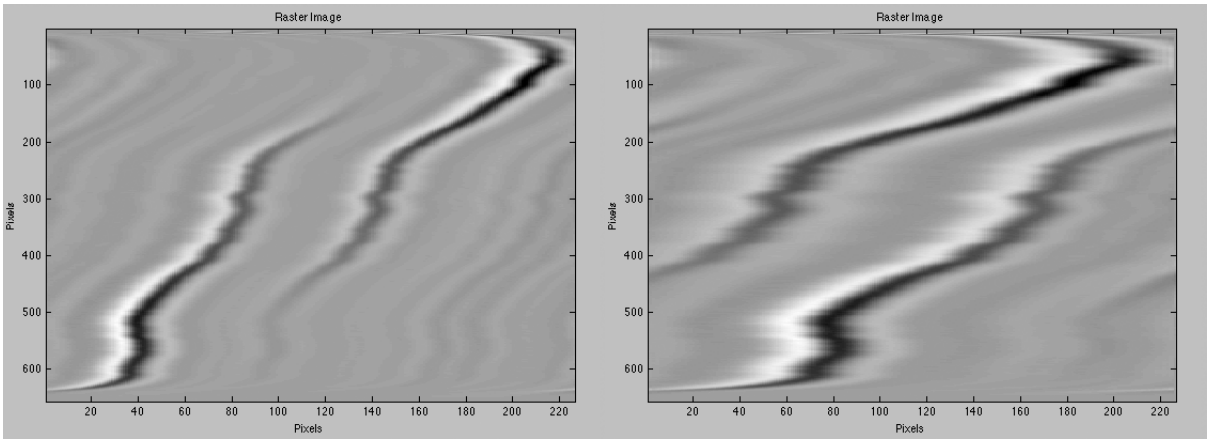
(c) Pitch Shifted Down 7 Semitones

Figure 6.15: Results of the Pitch Shift Process on a Trumpet Note in 1D Magnitude Spectrum

The interpolation also maintains a component at the original fundamental frequency causing an oscillation across the raster image width. The raster images of the pitch shifted signals shown in figures 6.15b and 6.15c are given in figure 6.16a, along with that of the unprocessed signal, to demonstrate the effects of this process on the time domain waveform.



(a) Original Waveform



(b) Pitch Shifted Up 5 Semitones

(c) Pitch Shifted Down 7 Semitones

Figure 6.16: Results of the Pitch Shift Process on a Trumpet Note in 1D Magnitude Spectrum

In rhythmic analysis mode the resolution of the audible frequency axis is much higher and the pitch of the signal can be changed. When the raster image width is defined as the smallest rhythmic duration in the signal, for example an eighth-note in simple120.wav, the pitch can be effectively altered without changing the tempo or rhythm of the signal. However if the raster image rows contain more than one note/event then the rhythm of the signal is also changed since the pitch adjustment is essentially changing the pitch and duration of each raster image row. The rectangular windowing of rasterisation also causes problems here since discontinuities are likely to occur at the joins of raster image rows.

In rhythmic analysis mode the interpolation also retains a component at the fundamental

frequency of analysis, causing an oscillation across the raster image rows. This effect is well demonstrated by the pitch shifting of loop120.wav in figure 6.17, and calls into question the appropriateness of resampling by interpolation.

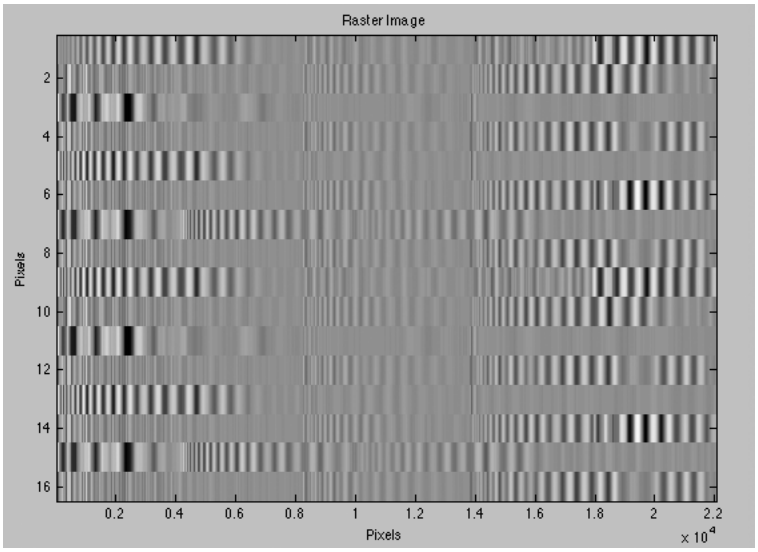


Figure 6.17: Raster Image of loop120.wav After Pitch Shifting

Even when the pitch shifting process produces acceptable results there are much more effective and efficient techniques available in 1D Fourier processing. It is still believed that there might be potential in effective pitch changing of signals using 2D Fourier processing, especially in timbral scale analysis where the rhythmic oscillations of harmonics could be adjusted independently from the audible frequencies to maintain an acceptable timbre (see section 6.3.1). However the resolution of analysis must be first be improved.

6.4.6 Rhythmic Frequency Range Adjustment

This process exhibits the same issues as pitch detection though based in the other frequency axis. When the rhythmic frequency range is altered by resampling, especially in rhythmic analysis mode, the frequency interval is too large and the tempo of the signal remains unchanged whilst the magnitudes of the original rhythmic frequencies are altered. This has potential for creative use, as does the timbral-mode pitch shifting, however it is not achieving the desired results.

Low frequency rhythmic variations are introduced when the rhythmic range is increased and since the DC rhythmic frequency row typically contains large magnitudes, this causes a signal drop out in magnitude across the duration of the signal as shown in figure 6.18 where the rhythmic range of loop120.wav was doubled.

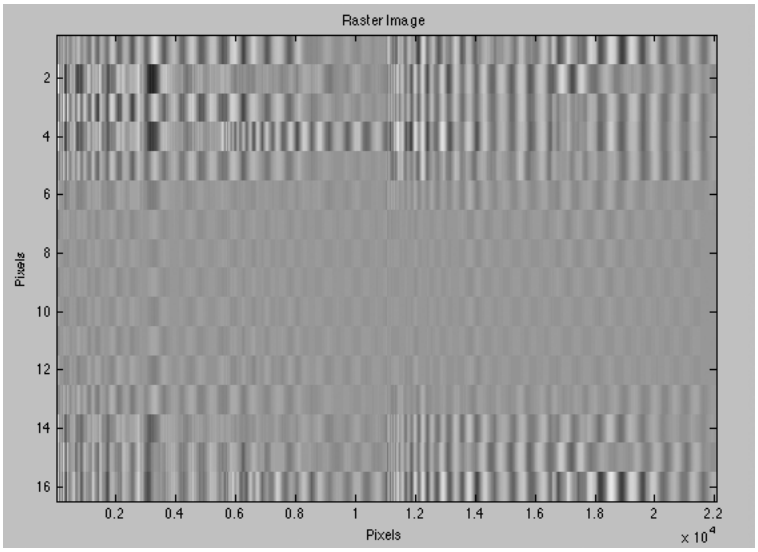


Figure 6.18: Raster Image of loop120.wav After Doubling Rhythmic Frequency Range

The same effects are observed in timbral analysis mode where the DC rhythmic frequency row is also generally large in magnitude. The fixed process 'Double Rhyth' that inserted zero-valued rows between each original row achieved the expected result without the errors caused by interpolation. This result is essentially increasing the tempo of the signal whilst maintaining the same duration, therefore the signal repeats itself. This is best demonstrated visually when processing a single instrument note as in figure 6.19.

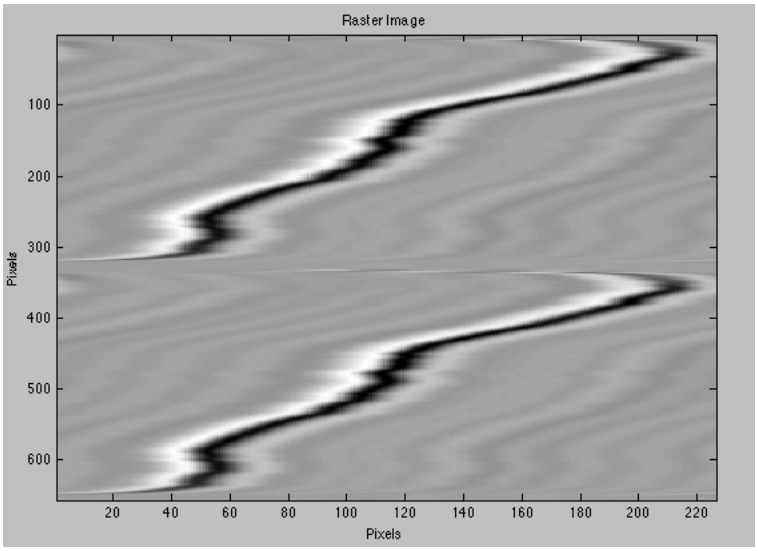
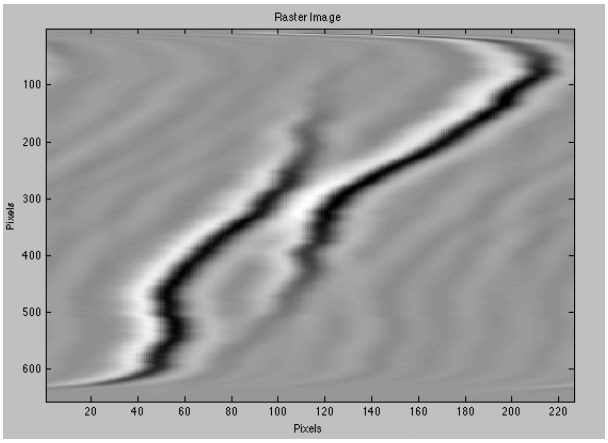
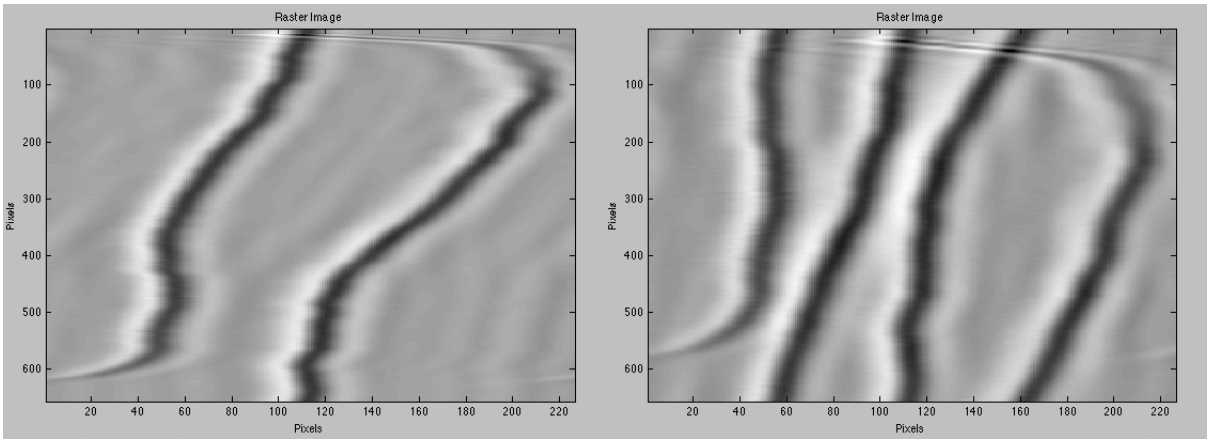


Figure 6.19: Trumpet Note Processed Using 'Double Rhyth'

Decreasing the rhythmic frequency range demonstrates an interesting result in timbral analysis mode where the rhythmic frequency resolution is larger. The signal duration is extended whilst maintaining the original pitch however since the raster image dimensions have not changed the duration of the data is unchanged therefore the signal restarts at the beginning of the image. Again this is well demonstrated by an instrument note signal, as shown in figure 6.20.



(a) Range Reduced By Factor of 0.75



(b) Range Reduced By Factor of 0.5

(c) Range Reduced By Factor of 0.25

Figure 6.20: Reducing the Rhythmic Frequency Range of a Trumpet Note

It is suggested that these issues stem from aliasing of rhythmic frequencies which seems unavoidable since the signal will always contain frequency components above half the rhythmic sampling frequency.

6.4.7 Resizing The 2D Fourier Spectrum

The resizing process does not suffer from the same limitations of resampling that the pitch shifting and rhythmic frequency range adjustment processes exhibit, since the analysis frequencies are being changed by the same factor as the signal frequencies. Interpolation still produces unwanted low-frequency variations when the frequency range is extended

in either dimension, although they are of lower magnitude when the dimensions of the spectrum are adjusted with the data.

The resizing of the spectrum width adjusts the tempo of the signal but also the pitch, since the audible frequencies of the signal are rescaled with the spectrum frequency points. Tempo adjustment without pitch change was attempted by the rhythmic frequency range adjustment. Resizing the height achieves a change in signal duration whilst maintaining the same tempo although it is limited by the same problems of aliasing as rhythmic frequency range adjustment.

6.4.8 Evaluation of Resampling Techniques

In retrospect the methods used to resample the spectrum were rudimentary. The poor results can partly be blamed on the frequency resolution of the 2D Fourier spectrum after raster scanning but the techniques and understanding of the processing could have been better informed. These initial prototype processes have still served a purpose however since it is now clearer what kind of signal transformations could potentially be performed and how the analysis process needs to be adapted to allow it.

6.4.9 Shifting Quadrants The 2D Fourier Spectrum

The quadrant shifting process does not produce very useful results. This is due to the disproportionate amount of low frequency energy in most musical audio signals which is shifted to the upper audible frequency range, at the upper frequency limit of human hearing. Figure 6.21 demonstrates the operation of this process by showing the result of applying it to loop120.wav. The resulting signals are barely audible and it is difficult to distinguish the expected change in rhythmic frequency characteristics as a result. The operation could be improved by combining downward pitch shifting with logarithmic rescaling of audible frequencies to get a more audible result.

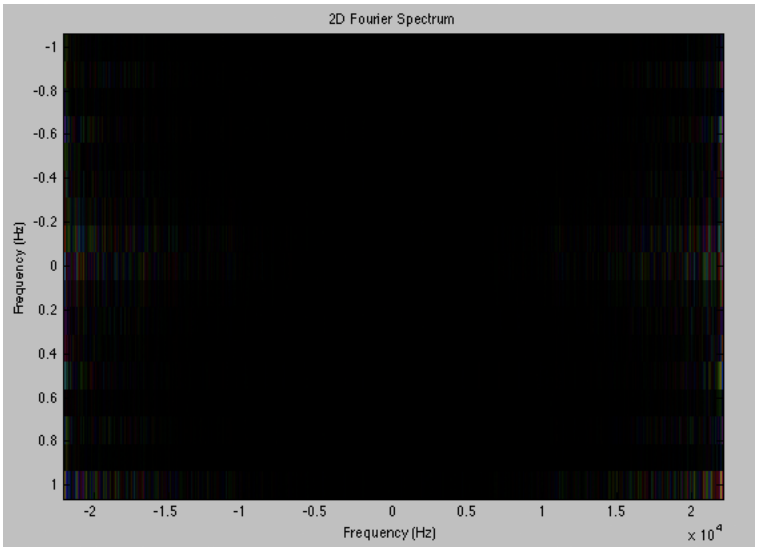


Figure 6.21: 2D Fourier Spectrum of loop120.wav After Quadrant Shifting

Chapter 7

Testing the Matlab Tool

Since this was an investigative project, the software development process was not clearly divided into phases of design, implementation and testing, but this does not mean that the design and testing was non-existent. Chapter 3 introduced the basic features of the software design and this chapter describes the testing processes used during the development of the 2D Fourier software tool.

The software tool has developed into a reasonably large application over the course of this investigation with many different functions and huge range of data paths and processing options. Whilst programming a constant effort was made to produce a robust and efficient design, however more importance was placed on achieving the aims of the project than exhaustively testing the software tool.

Over the course of the project, the roles of developer and user were carried out side by side. The software tool was constantly used to analyse and process audio, which not only allowed an understanding of the underlying signal processing to be developed but also revealed errors or unexpected processing results. The algorithms could then be examined using Matlab's debugging tools and corrections made or further testing pursued. Particularly complex or vulnerable areas of the program were tested with more formal methods, and several examples are provided in this chapter.

7.1 Black Box Testing

Many of the functions within the software tool perform specific manipulations of data arrays, either to convert between data representations or to transform the 2D spectrum data. Where appropriate, these functions were supplied with test data that would provide an expected output result. The actual result could then be compared with the expected result to confirm that the process performs the correct operation.

7.1.1 Testing `rasterise` and `derasterise`

The rasterisation process and its inverse are fundamental components of the project, see section 4.2. The `rasterise` function was written to convert from a 1D array to a 2D array using rasterisation. This process has a one-to-one sample mapping, so every item in the 1D array must be present in the 2D array and the 1D data stream should be readable in the manner shown in figure 2.5.

The following 1D 25 element array was input to the `rasterise` function to test that it performed the conversion corrrectly without losing data:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \dots & 24 & 25 \end{bmatrix}$$

The raster width was set to 5, so the expected output was:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

The output from `rasterise` was identical to this expected result. This 2D matrix was then used to test the `derasterise` function with a `hop` of 5. The 1D output of `derasterise` was identical to the original 1D array, showing that both functions operate correctly.

7.1.2 Testing `rev_fftshift`

The `rev_fftshift` function was tested to ensure that it correctly reassigned 2D spectrum quadrants to the original order in the `FT` array, performing the inverse operation to `fftshift` for an array of odd dimensions.

The test array with odd dimensions:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

Intermediate result after `fftshift`:

$$\begin{bmatrix} 19 & 20 & 16 & 17 & 18 \\ 24 & 25 & 21 & 22 & 23 \\ 4 & 5 & 1 & 2 & 3 \\ 9 & 10 & 6 & 7 & 8 \\ 14 & 15 & 11 & 12 & 13 \end{bmatrix}$$

Output result for `fftshift` on intermediate array:

$$\begin{bmatrix} 7 & 8 & 9 & 10 & 6 \\ 12 & 13 & 14 & 15 & 11 \\ 17 & 18 & 19 & 20 & 16 \\ 22 & 23 & 24 & 25 & 21 \\ 2 & 3 & 4 & 5 & 1 \end{bmatrix}$$

This demonstrates the requirement for the `rev_fftshift` function to correctly reassign the spectrum quadrants. The code for `rev_fftshift` is given in listing 7.1, each quadrant is shifted from the input array to the output using an array operation. Initially mistakes were made in the code when defining the quadrant dimensions which caused an assignment dimension mismatch error when the array operations were run. This meant that the code was corrected before the test was performed, but it highlighted the need to verify that the

correct process was being performed.

```
function out = rev_fftshift(in)
    y_size = size(in,1);
    x_size = size(in,2);
    y_cent = ceil(y_size/2);
    x_cent = ceil(x_size/2);
    out = in;
    % top left from bottom right
    out(1:y_cent,1:x_cent)=in(y_cent:y_size,x_cent:x_size);
    % bottom right from top left
    out((y_cent+1):y_size,(x_cent+1):x_size)=in(1:(y_cent-1),1:(x_cent-1));
    % top right from bottom left
    out(1:y_cent,(x_cent+1):x_size)=in(y_cent:y_size,1:(x_cent-1));
    % bottom left from top right
    out((y_cent+1):y_size,1:x_cent)=in(1:(y_cent-1),x_cent:x_size);
```

Listing 7.1: Reverse Fourier Spectrum Quandrant Shifting Function (`rev_fftshift`)

The intermediate array was input to `rev_fftshift` and the required result was the original test array.

Output result for `rev_fftshift` on intermediate array:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

7.2 Data Input Testing

Text edit objects are used many times within the software tool to allow the user to enter numerical parameters. This is a vulnerable point within the software since the user's actual input cannot be controlled.

The function `isnumber`, shown in listing 7.2, was written to check whether a text edit string

entered by the user could be converted to numerical data. It attempts to convert the input string to a numerical value using Matlab's `str2double` function. If the input string is not numeric then the result will have the value `NaN`, the IEEE arithmetic representation for Not-a-Number. Matlab's `isnan` function determines if a numeric variable has the value `NaN` and it is used check the ouput of the `str2double` function. The Boolean output of `isnan` is inverted to obtain the output of `isnumber`.

```
function is = isnumber(num)
    is = ~isnan(str2double(num));
    if ~is
        hMsg = errordlg([num ' is not numeric']);
        uiwait(hMsg);
    end
end
```

Listing 7.2: Checking Text Edit Input Is Numerical

If the text edit input is not numeric then the user is informed using an error dialogue window. Each text edit object in the software tool calls `isnumber` in its callback function. If the output value is true, then the function can proceed to use this numerical input as required and if it is false, the value of the underlying variable that is represented by the text edit object should be redisplayed.

7.3 Code Optimisation

The Matlab environment provides the Profiler tool to facilitate code optimisation. It is a GUI that allows Matlab code to be analysed using the `profile` function and displays the results. The Profiler tracks the execution time of code and presents information about number of calls, parent functions, child functions, code line hit count and code line execution time. The execution time of the code is increased by the Profiler but the ratio between execution times for each code line is accurate. The information obtained can be used to identify where code can be modified to achieve performance improvements.

The Profiler was used regularly throughout the software development. One example was to investigate the performance of the `rasterise` function. This function originally used

nested for loops to perform the rasterisation process by single-element operations as shown in listing 7.3, and its execution speed was too slow.

```
function image = rasterise(array, width)
    %calculate the required image size
    height = ceil(length(array)/width);
    %create an empty matrix
    image = zeros(height,width);
    for i=1:height
    for j=1:width
        arrayindex = ((i-1)*width)+j;
        if arrayindex <= length(array)
           image(i,j) = array(arrayindex);
        end
    end
end
```

Listing 7.3: Original Rasterisation Process

The `rasterise` function was run 10 times using a sinusoidal test signal, at a frequency of 220.5 Hz with a 1 second duration at a sampling rate of 44.1 kHz. The raster width was set to 200, the fundamental period of the sine wave.

The average total execution time was 1.205 seconds. Each code line within the two `for` loops was called for every sample of the input signal (44000 times) which led to a significantly long processing time.

The design of the function was reassessed to produce the implementation given in listing 4.1 using array operations instead of code loops. The new `rasterise` function was again run in the Profiler tool 10 times using the same sinusoidal test signal.

This time the average total execution time was 0.008 seconds, a significant reduction. The maximum code line hit count in this implementation was only 219, corresponding to the height of the image.

The testing in Profiler proved that data operations are much faster when performed using arrays than loops in Matlab. Array mathematics are inherent in the design of the Matlab environment whereas looped processes are slow because Matlab is an interpreted programming language (section 2.6). After this testing, array operations were used in favour of

code loops wherever possible. However conditional statements that compare a variable to a fixed value will not work with array variables since Matlab cannot support concurrent program paths. This therefore limits the use of array operations in certain instances.

7.3.1 YIN Pitch Algorithm Optimisation

Section 4.7.1 described the implementation of the YIN pitch algorithm in a MEX-file to obtain faster execution speeds than the original M-file implementation. The performance of MEX-file code cannot be analysed using the Profiler tool since it has to be pre-compiled from the C language source file. Matlab provides the `tic, toc` function which enables simple performance measurement using a stopwatch timer. The use of `tic, toc` is shown in listing 7.4. The function/code to be measured is inserted at the point '%any statements', and the variable `t` returns the elapsed time in seconds.

```
tic
    %any statements...
toc
t = toc
```

Listing 7.4: Use of `tic, toc` to Measure Code Performance

The two implementations of the YIN algorithm, `yinpitch` (M-file) and `yin` (MEX-file) were both run multiple times for a series of different audio signals. This is because the function performs a different number of operations depending how quickly the autocorrelation process identifies a value below the tolerance. Both functions used an input tolerance of 0.15, the recommended default in [3], which gave a satisfactory compromise between execution time and accuracy. Both functions were run 10 times for each audio signal and the results of this testing are shown in table 7.1.

File	Duration (seconds)	Pitch (Hz)	Execution Time (secs)		Speed Increase (%)
			M-File	MEX-File	
sine16.wav	0.9977	219.9556	0.1306	0.0194	673.2
square16.wav	0.9977	219.9501	0.1301	0.0196	663.7
am_bipolar.wav	2	219.9611	0.2588	0.0387	668.7
pianoC1.wav	2.8669	65.7846	1.2378	0.1824	678.6
clarinetC6.wav	2.0142	2018.3066	0.0309	0.0059	520.5

Table 7.1: Performance Comparison For Implementations of YIN Algorithm

The set of test signals was chosen to cover the common range of duration and pitch expected within the software tool, using both simple test signals and recordings of real instrument notes. The mean speed increase for all tests was 640.94% and this was quite consistent across the input signals with a standard deviation of 67.5526. It is thought that the algorithm found pitch of the file clarinetC6.wav very quickly, after few loop iterations, explaining the lower performance increase.

7.4 SNR of Analysis-Synthesis Process

It was necessary to investigate the signal-to-noise ratio of the 2D Fourier analysis-resynthesis process, as used in the software tool, to confirm that the perceptual quality of signal transformations would not be impaired by information loss in the analysis and synthesis operations.

Listing 7.5 shows the function `analysis_synthesis` which uses the software tools low-level analysis and synthesis functions. These functions are abstracted from the program data structure and settings. The function takes an audio array and the desired raster image width as input arguments and converts from the audio signal to the RGB colour representation of the 2D Fourier spectrum, and then back to the audio array. The colour display representation is included in the process since it the data cursor display is obtained from this array and its accuracy needed to be tested.

```
function audio = analysis_synthesis(audio,imwidth)
```

```
    image = rasterise(audio,imwidth);
    height_pad = mod(size(image,1),2);
    width_pad = mod(size(image,2),2);
    padded = padarray(image,[height_pad width_pad],0,'post');
    FT = fft2(padded);
    max_mag=calc_spec2D(FT,1,1,'magphase',true);
    spec2D=calc_spec2D(FT,1,1,'magphase',false);
    [magLN,phase]=colour2polar(spec2D,1,'magphase');
    magN =  2.^magLN - 1;
    mag = magN*max_mag;
    FT = rev_fftshift(complex(mag.*cos(phase),mag.*sin(phase)));
    image = real(ifft2(FT));
    actual_height = size(image,1) - height_pad;
    actual_width = size(image,2) - width_pad;
    image = image(1:actual_height,1:actual_width);
    audio = derasterise(image,imwidth);
end
```

Listing 7.5: The Software Tool's Underlying Analysis-Resynthesis Process

The function `process_SNR` was written to determine the signal-to-noise ratio of a reversible signal processing operation. The signal-to-noise ratio (in decibels) is given by the following equation:

$$\mathrm{SNR(dB)} = 20\log_{10}\left(\frac{\mathrm{A_{signal}}}{\mathrm{A_{noise}}}\right) \tag{7.1}$$

Where $\mathrm{A_{signal}}$ and $\mathrm{A_{noise}}$ are the root mean squared amplitudes.

The RMS amplitude of a signal x of length n is given by:

$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x(i) * x^*(i)} \tag{7.2}$$

Where $x^*(i)$ is the complex conjugate of $x(i)$.

The `process_SNR` function is shown in listing 7.6. It calculates the RMS amplitude of the input signal, then iterates the required signal processing operation the required number of times (given by the `iterations` argument). For each iteration of the operation, the noise

component is obtained as the difference between the original signal and the processed result after `i` iterations. The RMS amplitude of the noise and then the signal-to-noise ratio are calculated. The signal-to-noise ratio for each iteration of the process is stored in the array `SNR`, which is returned by `process_SNR` along with the `processed` signal.

```
function [SNR, processed] = process_SNR(original,iterations)

    rms_orig = sqrt(sum(sum(sum(original.*conj(original))))/size(original,1)
        );
    processed = original;
    for i = 1:iterations
        % HERE THE PROCESS IS PERFORMED
        processed = proc_function(processed);
        % get noise component
        noise = processed-original;
        rms_noise = sqrt(sum(sum(sum(noise.*conj(noise))))/size(noise,1));
        % calculate signal to noise ratio
        SNR(i) = 20*log10(rms_orig/rms_noise);
    end
end
```

Listing 7.6: Obtaining the Signal-To-Noise Ratio of a Process

The SNR of the analysis-synthesis process was tested by replacing `proc_function` with `analysis_synthesis`. The audio signal loop120mono.wav was used to test the SNR of the process. The raster image width was set to 22050, which is a quarter-note duration at the signal's tempo. This signal was chosen since it requires image padding in both dimensions and provides rich spectral content in both frequency dimensions.

The signal-to-noise ratio after the first iteration was 285.9285, and after the hundredth iteration it had only fallen to 279.1797. This can be compared to the SNR of quantisation error for 16-bit analogue-to-digital conversion of audio data, which is given by the following equation for a full-amplitude input signal:

$$\mathrm{SNR}_{\mathrm{ADC}} = 20\log_{10}\left(2^{Q}\right) dB \approx 96.33 dB \quad \text{where } Q = 16 \tag{7.3}$$

The signal-to-noise ratio of the analysis-synthesis process is therefore well within reasonable limits to allow processing in the 2D Fourier domain without undesirable information loss.

The `process_SNR` function was also used to investigate Matlab's `hsv2rgb` and `rgb2hsv` functions, revealing that a value compent of 0 in the HSV representation results in a loss of information. The range of the value component was therefore limited to [0.001 1] in the `polar2colour` function, as described in section 4.4.1.

Chapter 8

Conclusions

A comprehensive software tool has been developed in Matlab to allow 2D Fourier analysis of audio signals. It uses the rasterisation process to obtain a 2D time-domain representation of the signal which can then be converted to the frequency domain using the 2D Fourier transform. The investigation has led to an understanding of signal properties in the 2D frequency domain. Based on this understanding, experiments were performed into audio transformation processes that manipulate the 2D Fourier domain representation and these were incorporated into the software.

The rasterisation process applies a rectangular window to an audio signal to divide it into evenly sized rows with no overlap. These rows can then be aligned and the signal can be displayed in two dimensions as an image. When the row width is correctly set according to a signal periodicity the raster image can display slow variations of the waveform characteristics in the vertical axis.

The raster image and the 2D Fourier spectrum are displayed visually in the software tool along with the 1D time and frequency domain representations. A novel method of displaying the complete 2D spectrum data has been implemented by converting from a polar representation to a colour value, where brightness corresponds to magnitude and hue corresponds to phase. This process incorporates brightness and contrast parameters to allow the range and scaling of the data to be adjusted, allowing a flexible investigation of the signal. Matlab's plot tools are utilised to provide a detailed analysis environment.

The fundamental component of the 2D Fourier spectrum can be considered as a sine wave

with a stationary audible frequency and a stationary sub-sonic frequency of rhythmic variation, which is an amplitude modulated sine wave. When the analysis frequency points of the 2D spectrum are precisely synchronised with the audible and rhythmic frequency of this component, it is represented by four distinct points symmetrically placed in each quadrant of the spectrum. These points do not have identical magnitude values and corresponding opposite phase as with the two points that represent a single sinusoidal component in the 1D Fourier domain representation. This is because the 2D Fourier transform is implemented by two sequential DFT processes on the same audio data, the second one is operating on the complex result of the first. The complex DFT does not produce symmetrically valued results but it is thought that the relationship between the positive and negative frequency points is linear.

When the analysis of a signal component is not synchronised with its audible or rhythmic frequency, the signal is displayed at an angle in the raster image and the corresponding 2D spectrum points are skewed according to this angle. The spectral energy of this component is also smeared between adjacent points because it's audible and rhythmic frequencies do not precisely match the analysis frequencies. This reduces the clarity of the 2D Fourier spectrum representation.

Concise 2D spectral analysis, with less spectral smearing of frequency components, is achieved when the raster image width is set according to analysis of either pitch or tempo. When the raster image width is synchronised according to the pitch of the signal, the audible frequency content is well defined since frequency bins correspond to signal harmonics. When the raster image width is synchronised according to the tempo of the signal, the rhythmic frequency variation is well defined since frequency bins correspond to prominent rhythmic variations at the signal's tempo.

The sample rate is a limiting factor when attempting to synchronise audible or rhythmic frequencies of the 2D spectrum components, since the period length rarely corresponds to an integer number of samples. Resampling should have been performed before the 2D Fourier transform to ensure synchronised and accurate analysis.

The 2D Fourier domain processing experiments performed in this project have shown mixed results. Filtering and magnitude thresholding of the 2D spectrum data provides unique and musically useful results when applied to the rhythmic frequency dimension, altering the sub-sonic structure of the signal without affecting its audible frequency range. In

the audible frequency dimension, these processes operate in a similar manner to their 1D equivalent.

Time/pitch modifications of the signal that involve resampling of the spectrum data have shown poor results. This is due partly to the analysis process which was not designed to ensure these processes were possible. The resolution of the analysis frequencies is generally too low to allow accurate resampling of spectrum data. It is believed that there may still be potential for independent rescaling of rhythmic and audible frequencies to produce useful results, especially when the rhythmic oscillations of signal harmonics are adjusted after pitch change. It appears that there are also issues with the aliasing of rhythmic frequencies, which may be unavoidable.

The use of rasterisation in 2D Fourier analysis is essentially a restricted implementation of Penrose's method [23], which extends the STFT into two frequency dimensions, applying a 2D window function. Rasterisation performs this process with a rectangular 2D window and a hop size equal to the size of each frame. By removing the specific hop size requirement of rasterisation and overlapping the analysis frames, the frequency resolution of both axes could be increased. A tapered window such as a 2D Hamming window would have to be used, but this would also reduce the spectral smearing of unsynchronised signal components.

In summary, the 2D Fourier transform offers an exciting new perspective on audio signal analysis and processing. It suffers from various limitations as do most signal processing paradigms however it certainly has a lot of potential. Rasterisation provides a useful visual display of low frequency waveform variations and has greatly helped the understanding of 2D Fourier domain signal properties. However it applies unnecessary restrictions on the 2D Fourier analysis and for further work it is recommended that a variable hop size is used.

Chapter 9

Future Work

Suggestions for future work:

- Variable hop size should be implemented to allow frame overlap with the use of 2D signal windowing, since the analysis resolution can be increased and processing algorithms may be improved.
- 2D Fourier analysis and processing methods need deeper theoretical investigation so that the techniques initially investigated in this project can be improved based on a clearer understanding.
- Timbral analysis - An instrument/instrument family has similar 2D spectral envelope for all notes, the 2D Fourier transform can be extremely useful for defining instrument timbres.
- Component extraction - the extra rhythmic frequency information may possibly facilitate source separation methods.
- Transformations - There are surely many more 2D spectral audio transformations possible. For example, the manipulation of phase information was not investigated. Truly unique techniques may be obtainable by performing operations that alter both rhythmic and audible frequencies, such as obtaining the inverse matrix (although this has very specific requirements).
- The software tool could be developed as a stand-alone or offline VST application, independent of MATLAB, to make the techniques more broadly accessible to engineers

and composers.

Bibliography

[1] M. K. Agoston. *Computer Graphics and Geometric Modeling : Implementation and Algorithms*. Springer-Verlag, 2005.

[2] S. Bernsee. Time stretching and pitch shifting of audio signals. http://www.dspdimension.com/admin/time-pitch-overview/.

[3] P. Brossier. Aubio - a library for audio labelling. http://aubio.org/.

[4] D. Byrd. Midi note number to equal temperament semitone to hertz conversion table. Indiana University Bloomington School of Music. http://www.indiana.edu/ emusic/hertz.htm.

[5] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[6] K. Cowtan. Picture book of fourier transforms. http://www.ysbl.york.ac.uk/~cowtan/fourier/fourier.html.

[7] A. de Cheveigné and H. Kawahara. Yin, a fundamental frequency estimator for speech and music. *Journal of the Acoustical Society of America*, 111(4):1917–1930, 2002.

[8] A. De Götzen, N. Bernardini, and D. Arfib. Traditional (?) implementations of the phase vocoder. In *Proceedings of the 3rd International Conference on Digital Audio Effects*, 2000.

[9] M. Dolson. The phase vocoder: A tutorial. *Computer Music Journal*, 10(4):14–27, 1986.

[10] T. Erbe. *SoundHack User's Manual*. School of Music, CalArts.

[11] K. Fishkin. A fast hsl-to-rgb transform. In A. S. Glassner, editor, *Graphics Gems*,

pages 448–449. Academic Press, 1998.

[12] K. Fitz and L. Haken. On the use of time-frequency reassignment in additive sound modeling. *Journal of the Audio Engineering Society*, 50(11):879–893, 2001.

[13] J. L. Flanagan and R. M. Golden. The phase vocoder. *The Bell System Technical Journal*, 45(8):1493–1509, 1966.

[14] D. Gabor. Acoustical quanta and the theory of hearing. *Nature*, 159:591–594, 1947.

[15] J. Gallicchio. 2d fft java applet. http://www.brainflux.org/java/classes/FFT2DApplet.html.

[16] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Prentice Hall, 3rd edition, 2008.

[17] R. Kirk and A. Hunt. *Digital Sound Processing for Music & Multimedia*. Focal Press, 1999.

[18] J. Laroche and M. Dolson. New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects. *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 91–94, 1999.

[19] O. Lartillot and P. Toiviainen. A matlab toolbox for musical feature extraction from audio. In *Proceedings of the 10th International Conference on Digital Audio Effects*, 2007.

[20] O. Lartillot, P. Toiviainen, and T. Eerola. Mirtoolbox. Department of Music, University of Jyväskylä. http://www.jyu.fi/hum/laitokset/musiikki/en/research/coe/materials/mirtoolbox.

[21] R. McAulay and T. Quatieri. Speech analysis/synthesis based on a sinusoidal representation. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(4):744–754, 1086.

[22] J. A. Moorer. The use of the phase vocoder in computer music applications. *Journal of the Audio Engineering Society*, 26(1/2):42–45, 1978.

[23] C. Penrose. Chapter 2: Spectral representations. Taken from incomplete thesis at http://silvertone.princeton.edu/~penrose/thesis/, 2008.

[24] J. O. Pickles. *An Introduction to the Physiology of Hearing*. Academic Press, 2nd edition, 1988.

[25] W. Pratt. *Digital Image Processing*. Wiley-Interscience, 2nd edition, 1991.

[26] C. Roads. Pitch & rhythm recognition. In *The Computer Music Tutorial*, chapter 12. MIT Press, 1995.

[27] C. Roads. *Microsound*. MIT Press, 2001.

[28] E. A. Robinson. A historical perspective of spectrum estimation. In *Proceedings of the IEEE*, volume 70, pages 885–907, September 1982.

[29] M.-H. Serra. Introducing the phase vocoder. In C. Roads, S. T. Pope, A. Piccialli, and G. D. Poli, editors, *Musical Signal Processing*. Swets & Zeitlinger, 1997.

[30] X. Serra and J. O. Smith. Spectral modelling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal*, 14(4):12–24, 1990.

[31] S. W. Smith. *The Scientist and Engineers' Guide to Digital Signal Processing*. California Technical Pub., 1st edition, 1997.

[32] I. Y. Soon and S. N. Koh. Speech enhancement using 2-d fourier transform. *IEEE Transactions on Speech and Audio Processing*, 11(6):717–724, 2003.

[33] T. Tolonen, V. Välimäki, and M. Karjalainen. Evaluation of modern sound synthesis methods. Technical Report 48, Helsinki Institute of Technology, March 1998.

[34] H. Valbret, E. Moulines, and J. P. Tubach. Voice transformation using psola technique. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 145–148, San Francisco, CA, USA, 1992.

[35] W3C. Css level 3 color module specification. http://w3.org/TR/css3-color/.

[36] J. J. Wells. *Real-Time Spectral Modelling of Audio for Creative Sound Transformation*. PhD thesis, University of York, 2006.

[37] J. J. Wells. Writing mex-files for matlab. MA/MSc/Diploma Music Technology Audio Processing Techniques and Environments Laboratory Script, University of York, 2008.

[38] W. S. Yeo and J. Berger. A framework for designing image sonification methods. In *Proceedings of 11th International Conference on Auditory Display*, Limerick, Ireland, July 2005.

[39] W. S. Yeo and J. Berger. Application of raster scanning method to image sonifi-

cation, sound visualization, sound analysis and synthesis. In *Proceedings of the 9th International Conference on Digital Audio Effects*, Montreal, Canada, 2006.

Appendices

Appendix A

Correspondence With External Academics

All correspondence with academics other than the two project supervisors is documented here.

A.1 Complex Colour Representation

The process of converting between a polar data representation to an RGB colour value was based on the work by Jason Gallicchio, a graduate physics student at Harvard. His personal website [15] contains a Java applet demonstrating the properties of the 2D Fourier transform of images which used displayed Fourier data using a colour representation. Gallicchio was contacted regarding this technique and his reply is reproduced in this section.

A.1.1 Email Correspondence

Here is the e-mail correspondence with Jason Gallicchio about his work in [15] on conversion from complex data to a colour representation.

From: jason@physics.harvard.edu
Subject: Re: Complex to colour conversion
Date: 27 February 2008 07:00:03 GMT

To: cwp500@york.ac.uk

The basic idea is to map the phase to hue and magnitude to lightness in HSL color space. Unfortunately Java has a routine only for HSV, so we have to calculate the saturation and value.

http://en.wikipedia.org/wiki/HSV_color_space

Annoying Note: Java calls it HSB where Java's brightness is Wikiepedia's value even though Wikipedia says brightness is synonymous with lightness rather than value.

It might help to look at the image of the HSV cylinder. Magnitudes less than R are fully saturated and their value decreases to become blacker. Magnitudes greater than R have full value, but their saturation decreases to make them whiter. R is adjusted by the slider. The other variables (bmax, bmin, lmax, lmin) never change and keep their simple 1 or 0 values, so I should really optimize them out. To map an infinite range of numbers (0 to infinity) to a finite interval, the arc tangent function is very useful (or arccot):

http://commons.wikimedia.org/wiki/Image:Atan_acot_plot.svg

As for mapping the phase to hue, getting the angle from the (x,y) coordinates is perfect for the atan2 function, which keeps track of which quadrant you're in. Taking atan(y/x) doesn't work because if both are negative, for example, you get the same result as when both are positive. For some reason I had to add 2pi or pure-real numbers didn't work:

$$phi = 2.0 * Math.PI + Math.atan2(im, re)$$

The only other complication with the hue was that going from angle directly to hue spends too much time on red, green, and blue, and transitions through yellow, cyan, and magenta too fast. I added some small amount of a sin going at 3x the speed so that the hue spends more time in the transition regions to make the map smoother (compare the Wikipedia HSV cylinder to the rainbow button on the 2D FFT Applet)

$$hue = phi + Math.sin(3.0 * phi)/5$$

This description has been for the MAG_PHASE mapping. When you look at only the real, imaginary, magnitude, or phase info, you can simplify the mapping. When looking at just the magnitude, I go straight to grayscale. When look at just the phase, I calculate the hue as before, but set the saturation and brightness to full.

I pasted this email as a big comment on the top of the ComplexColor.java file, which you can rename ComplexColour if you really must. :-)

What do you mean by 2D Fourier representation of audio? You must mean something like taking a second at a time and calculating the 1D FFT for just that second, then taking the next second. There's a name for this kind of plot, but I can't think of it. It's NOT a 2D FFT, though. Here's an example where rainbow colors map to magnitude and phase info is neglected:

http://note.sonots.com/?SciSoftware%2Fstft

Jason

Chris Pike wrote:
Hi Jason,
I'm doing my MEng project at the moment and it involves 2D Fourier representation of audio. Your applet has been really helpful whilst trying to get my head around the 2D Fourier transform. I was wondering if you could help me by explaining the algorithm for converting from a complex number to an RGB colour? I've looked at your code in ComplexColour.java but I'm not entirely sure why it works or what the variables represent. I appreciate that you might not have time to help, if you have any references you could give that would also be great help. Thanks,
Chris Pike

A.1.2 ComplexColor Java Class

Listing `complexcolour` shows the Jason Gallicchio's ComplexColor Java class, which is referred to in the above email and was used as the basis for the `polar2colour` and `colour2polar` functions written for the software tool.

```
package brainflux.graphics;

import java.awt.Color;

/**
 * Title:        FFT2D
 * Description:
```

```
 * Copyright:     Copyright (c) 2001
 * Company:       BrainFlux
 * @author Jason Gallicchio
 * @version 1.0
 */

public class ComplexColor {

  private float R=1.0f, bmin=0.0f, bmax=1.0f, lmin=0.0f, lmax=1.0f;
  // Drawing options.  These are the same constants defined in brainflux.
     math.Complex.
  public static final int MAG_PHASE  = 0;
  public static final int MAG        = 1;
  public static final int PHASE      = 2;
  public static final int REAL       = 3;
  public static final int IMAG       = 4;
  public static final int MAG2       = 5;
  public static final int MAG2_PHASE = 6;

  private int type = MAG_PHASE;  // One of the above constants.

  public ComplexColor() {
  }

  public ComplexColor(float R) {
    this.R = R;
  }

  public ComplexColor(float R, float bmin, float bmax,
                      float lmin, float lmax, int type) {
    set_constants(R, bmin, bmax, lmin, lmax, type);
  }

  public void set_constants(float R, float bmin, float bmax,
                            float lmin, float lmax, int type)
  {
    this.R=R;
    this.bmin=bmin;
    this.bmax=bmax;
    this.lmin=lmin;
```

```
    this.lmax=lmax;
    this.type=type;
  }

  public void setR(float R) {
    this.R = R;
  }

  public float getR() {
    return R;
  }

  public void setDrawingType(int type) {
    this.type = type;
  }

  public int complex_to_color(float re, float im) {
    if (type==REAL)
      im = 0;
    if (type==IMAG)
      re = 0;
    float hue, sat, brightness;
    float r;
    if (type==PHASE)
      r = R;
    else {
      r = re * re + im * im;  // For now, r^2
      if (type != MAG2 || type != MAG2_PHASE)
        r = (float) Math.sqrt(r);
    }
    float li = Math.min(1.0f, (Math.max(0.0f,
            (float)Math.atan(r/R) * 2.0f/(float)Math.PI)*(bmax-bmin) + bmin)
                ) *
            (lmax-lmin)+lmin;
    if (type==MAG) {
      int dark = (int)(li*256f);
      return 0xFF000000+(dark<<16)+(dark<<8)+(dark);
    }
    if (li<0.5f) {
      sat = 1.0f;
```

```
      brightness = 2f*li;
    }
    else {
      sat = 2.0f-2.0f*li;
      brightness = 1.0f;
    }
    //brightness = ((float)Math.atan(r/R) * 2f/(float)Math.PI);
    float phi = (float)Math.PI+(float)Math.atan2(im,re);
    double scale = (double)4.0/20.0;
    hue = (float)( phi+scale*Math.sin(3*phi) ) / (float)(2*Math.PI);  //
        Correct for the "banding" in HSB color where RGB are wider than the
        others
    int rgb = Color.HSBtoRGB(hue, sat, brightness);
    return rgb;
  }
}
```

Listing A.1: Gallicchio's ComplexColor Class (http://www.brainflux.org/java.tar.bz2)

A.2 Two-Dimensional Fourier Processing of Audio

This section reproduces the e-mail correspondence with Christopher Penrose about 2D Fourier processing of audio. Chapter 2 of Penrose's incomplete thesis [23], is available on his personal website. It was this work that first suggested the use of the 2D Fourier transform to adjust sub-sonic frequency content of audio. Penrose discussed the developments of his work on 2D Fourier processing in the e-mail shown below.

Hi Chris!

I have written a few different sound processors that use 2d Fourier transforms. They were developed for the UNIX shell. Unfortunately, most of my 2d processors were written using Apple's veclib so they aren't very portable. But your screenshots on your website make it look like you are using OSX yourself. Some of these processes would have been documented in my thesis, but I still haven't finished that... =) I am working on a starting a music software company at the moment and I am getting its first application ready for release. And there is a long line behind that application already formed.... bad news for

my thesis! =)

I really believe that 2D Fourier transforms have interesting potential, particularly when frequency independent operations are performed. I have a few different adaptive "rhythmic" filtering processes which have turned out to be compelling sound permutation processes. Focused "rhythmic" filtering (resynthesizing a 2D DFT with a few narrow bands significantly amplified) is even somewhat interesting, albeit obvious, as a processing technique. Since I have only done this in the simplistic manner that I have described, the time granularity I have achieved is limited. But it may be possible to obtain similar effects with finer granularity than I achieved.

Personally, I am a strong advocate for studio sound processing techniques but unfortunately the marketability of processors that utilize 2D FFTs is limited by their intrinsic non-real-time architecture. They wouldn't play well in even the most liberal audio plug-in apis. I am glad that I can use my 2d processes easily in a UNIX shell however. =) And MATLAB is a decent home for them as well.

I will try to yoho a copy of MATLAB so I can hear what you have been up to =)

My company's first commercial app is a graphical sound synthesis and scoring system inspired by the Upic system. It is in the late stages of development; particularly since I had a working prototype in 1992 and have been honing its dsp engine for nearly 16 years =) It will be named, Optasia; it is derived from my freeware application Hyperupic which I originally developed for NeXT computers. Fortunate for me, MacOS X has embraced the same application development API that I used in the early 90s on a NeXT. I will even be able to easily port to the iPhone and iPod touch.

And thank you for reminding me about this! It helped me rescue a program from obscurity which I wrote a few years ago using 2D DFTs .

Best,

Christopher

On May 21, 2008, at 5:27 AM, Chris Pike wrote:

Hello, I'm glad to have found you. I've only read the two chapters of your thesis that are on your web page but my MEng project has been based on the idea of two-dimensional Fourier analysis that you suggest in Chapter 2. I'm finishing it up now but I've been working in

MATLAB to create a tool that allows a user to load audio and view its 2D Fourier spectrum, as well as apply several transformations to the spectrum data and resynthesize the audio. I'm using raster scanning (http://ccrma.stanford.edu/∼woony/works/raster/) to obtain a 2D representation with no frame overlap, since this was my project brief.

Anyway, I was just keen to know how far you took this idea of manipulating rhythmic frequency and whether you think it has much potential. From the work I've done the transformations seem quite unusual and interesting but to the extent I've developed it, perhaps a bit limited. It seems like there's also scope for this rhythmic-audible frequency grid to be used for in depth audio analysis. I'd really like to hear what you have to say on the matter since I've been working for nearly 5 months on this and not really had any reference point to other research, I've occasionally felt a bit out of my depth.

I've got a web page at http://www-users.york.ac.uk/∼cwp500/. There's not much information on there and the sound files are from my earliest investigations but there is a ZIP file containing my MATLAB code if you're interested.

Thanks, Chris

On 21 May 2008, at 08:19, Christopher Penrose wrote:

Hi Chris

Greg Smith forwarded your inquiry about my thesis to me. I have to look into the broken forwarding on my music.princeton.edu address. Fire away any questions you may have.

Best,

Christopher

Appendix B

Accompanying Compact Disc

The accompanying compact disc provides an electronic copy of this report in PDF format, as well as a the Matlab code for the 2D Fourier software tool and a folder of audio examples that demonstrate the results of 2D Fourier domain processing.

The software tool can be run by setting Matlab's workspace directory to the 'Source' folder containing the M-file code and then entering `app` in the command line. Please note that in order to use the tempo calculation function, the MIRtoolbox must be downloaded and installed [20]. The software was development on an Apple computer so whilst the M-file functions are system independent, the YIN algorithm must be recompiled as a MEX-file from `yin.c` on the system on which it will be run. Matlab provides detailed documentation on this process. The GUI display may also be slightly different on a Windows operating system and hence will not precisely match the images in the report.

The audio examples are grouped in folders according to the transformation process that was applied. The files are listed here, by each folder within 'Audio Examples', along with the process parameters that were used.

Unprocessed

These audio files are the original unprocessed signals that have been used in these examples, as well as signals displayed in the report as analysis examples. They should be used as a reference to demonstrate the effects of the 2D Fourier domain transformations.

- `am_bipolar.wav`

  This is an amplitude modulated sine wave synthesised in Matlab with a carrier fre-

quency of 220.5 Hz and a modulation frequency of 1 Hz.

- `loop120.wav`
  An electronic drum beat with complex rhythmic content at a tempo of 120 bpm and with a duration of 4 bars. Taken from the PowerFX sample pack (http://www.powerfx.com).
- `Piano.mf.A3.wav`
  A short extract of a piano recording taken from The University of Iowa Musical Instrument Samples (http://theremin.music.uiowa.edu/MIS.html). It has the dynamic ***mf*** and the note A3.
- `Piano.mf.A4.wav`
  A short extract of a piano recording taken from The University of Iowa Musical Instrument Samples. It has the dynamic ***mf*** and the note A4.
- `Piano.mf.C1.wav`
  A short extract of a piano recording taken from The University of Iowa Musical Instrument Samples. It has the dynamic ***mf*** and the note C1.
- `Piano.mf.F2.wav`
  A short extract of a piano recording taken from The University of Iowa Musical Instrument Samples. It has the dynamic ***mf*** and the note F2.
- `simple120.wav`
  A 4 bar drum rhythm programmed in a MIDI sequencer using Native Instruments Battery 3's 'Basic Kit' drum samples.
- `sine16.wav`
  A sine wave at 220.5 Hz, synthesised in Matlab.
- `trumpetG3.wav`
  A short extract of a B^b trumpet recording taken from The University of Iowa Musical Instrument Samples. It has the dynamic ***mf*** and the note G3.
- `TrumpetG4.wav`
  A short extract of a B^b trumpet recording taken from The University of Iowa Musical Instrument Samples. It has the dynamic ***mf*** and the note G4.
- `Violin.arco.mf.sulG.G3.wav`
  A short extract of a violin recording taken from The University of Iowa Musical

Instrument Samples. It has the dynamic ***mf***, and the note G3 played arco on the G string.

- `Violin.arco.mf.sulG.G4.wav`
  A short extract of a violin recording taken from The University of Iowa Musical Instrument Samples. It has the dynamic ***mf***, and the note G4 played arco on the G string.

Column Shift

These audio files have been processed using the column shift transformation.

- `simple120_eighthwidth_shift50_leave.wav`
  - Shift: 50
  - Option: Leave original rows
- `simple120_eighthwidth_shift50_wrap.wav`
  - Shift: 50
  - Option: Wrap rows around
- `simple120_eighthwidth_shift280_leave.wav`
  - Shift: 280
  - Option: Leave original rows
- `simple120_eighthwidth_shift280_remove.wav`
  - Shift: 280
  - Option: Remove original rows
- `trumpetG3_shift5_remove.wav`
  - Shift: 5
  - Option: Remove original rows
- `trumpetG3_shift7_remove.wav`
  - Shift: 7
  - Option: Leave original rows

Filtering

These audio files have been processed using the 2D spectral filter.

- `loop120_aud_1D_LP2kHz_ideal.wav`

  This example has been processed using ideal 1D Fourier low-pass filtering at 2kHz to serve as a comparison to the 2D equivalent.

- `loop120_aud_1D_LP2kHz_ideal.wav`
  - Source file: loop120.wav
  - Raster Image Width: One quarter-note
  - Frequency Mode: Audible
  - Type: Low-pass
  - Cutoff: 2 kHz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `loop120_aud_quartwidth_LP_750Hz_but02_cut.wav`
  - Source file: loop120.wav
  - Raster Image Width: One quarter-note
  - Frequency Mode: Audible
  - Type: Low-pass
  - Cutoff: 750 Hz
  - Frequency response: 2^{nd} Order Butterworth
  - Cut/boost: Cut
- `loop120_aud_quartwidth_LP_750Hz_ideal_cut.wav`
  - Source file: loop120.wav
  - Raster Image Width: One quarter-note
  - Frequency Mode: Audible
  - Type: Low-pass

  - Cutoff: 750 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `loop120_eighthwidth_BP_2Hz_pt125_but02_boost.wav`
  - Source file: loop120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic
  - Type: Band-pass
  - Cutoff: 2 Hz
  - Bandwidth: 0.125 Hz
  - Frequency response: 2^{nd} Order Butterworth
  - Cut/boost: Boost
- `loop120_eighthwidth_HP_1Hz_ideal_cut_DCrhyth.wav`
  - Source file: loop120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic
  - Type: High-pass
  - Cutoff: 1 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
  - DC rhythmic frequency row added as a pass band
- `loop120_eighthwidth_HP_1Hz_ideal_cut.wav`
  - Source file: loop120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic

  - Type: High-pass
  - Cutoff: 1 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `loop120_eighthwidth_LP_0pt5Hz_but04_cut.wav`
  - Source file: loop120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic
  - Type: Low-pass
  - Cutoff: 0.5 Hz
  - Frequency response: 2^{nd} Order Butterworth
  - Cut/boost: Cut
- `loop120_quarterwidth_BS_0Hz_pt125_ideal_cut.wav` This filter removes the DC rhythmic frequency row.
  - Source file: loop120.wav
  - Raster Image Width: One quarter-note
  - Frequency Mode: Rhythmic
  - Type: Band-stop
  - Cutoff: 0 Hz
  - Bandwidth: 0.125 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `loop120_quartwidth_LP_0pt5Hz_but04_cut.wav`
  - Source file: loop120.wav
  - Raster Image Width: One quarter-note
  - Frequency Mode: Rhythmic

  - Type: Low-pass
  - Cutoff: 0.5 Hz
  - Frequency response: 4^{th} Order Butterworth
  - Cut/boost: Cut
- `simple120_eighthwidth_BP_1Hz_pt125bw.wav`
  - Source file: simple120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic
  - Type: Band-pass
  - Cutoff: 1 Hz
  - Bandwidth: 0.125 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `simple120_eighthwidth_HP_1pt5Hz_ideal.wav`
  - Source file: simple120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic
  - Type: High-pass
  - Cutoff: 1.5 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `simple120_eighthwidth_LP_1Hz_ideal_cut.wav`
  - Source file: simple120.wav
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic

  - Type: Low-pass
  - Cutoff: 1 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `trumpetG3_aud_BP_but02_800Hz_bw400_cut.wav`
  - Source file: trumpetG3.wav
  - Frequency Mode: Audible
  - Type: Band-pass
  - Cutoff: 800 Hz
  - Bandwidth: 400 Hz
  - Frequency response: 2^{nd} Order Butterworth
  - Cut/boost: Cut
- `trumpetG3_BP_but02_8Hz_4bw_cut.wav`
  - Source file: trumpetG3.wav
  - Frequency Mode: Rhythmic
  - Type: Band-pass
  - Cutoff: 8 Hz
  - Bandwidth: 4 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `trumpetG3_BP_ideal_0Hz_cut.wav`
  - Source file: trumpetG3.wav
  - Frequency Mode: Rhythmic
  - Type: Band-pass
  - Cutoff: 0 Hz

  - Bandwidth: 0.5923 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut
- `trumpetG3_LP_but02_5Hz_cut.wav`
  - Source file: trumpetG3.wav
  - Frequency Mode: Rhythmic
  - Type: Low-pass
  - Cutoff: 5 Hz
  - Frequency response: 2^{nd} Order Butterworth
  - Cut/boost: Cut
- `trumpetG3_LP_but02_30Hz_cut.wav`
  - Source file: trumpetG3.wav
  - Frequency Mode: Rhythmic
  - Type: Low-pass
  - Cutoff: 30 Hz
  - Frequency response: 2^{nd} Order Butterworth
  - Cut/boost: Cut
- `trumpetG3_LP_ideal_30Hz_cut.wav`
  - Source file: trumpetG3.wav
  - Frequency Mode: Rhythmic
  - Type: Low-pass
  - Cutoff: 30 Hz
  - Frequency response: Ideal
  - Cut/boost: Cut

Fixed

These audio files demonstrate the results of the fixed transformation processes for `simple120.wav`

and `trumpetG3.wav`, in rhythmic and timbral analysis mode respectively. An eighth-note row duration was used for `simple120.wav`.

- `loop120_aud_1D_LP2kHz_ideal.wav`
  This example has been processed using ideal 1D Fourier low-pass filtering at 2kHz to serve as a comparison to the 2D equivalent.
- `simple120_eightwidth_doubledur.wav`
  - Process: Double Duration
- `simple120_eightwidth_doublerhyth.wav`
  - Process: Double Rhythm
- `simple120_eightwidth_doubletempo.wav`
  - Process: Double Tempo
- `simple120_eightwidth_downoctave.wav`
  - Process: Down Octave
- `simple120_eightwidth_halvedur.wav`
  - Process: Halve Duration
- `simple120_eightwidth_halverhyh.wav`
  - Process: Halve Rhythm
- `simple120_eightwidth_halvetempo.wav`
  - Process: Halve Tempo
- `simple120_eightwidth_specshift.wav`
  - Process: Spectrum Quadrant Shift
- `simple120_eightwidth_upoctave.wav`
  - Process: Up Octave
- `trumpetG3_doublerhyth.wav`
  - Process: Double Rhythm
- `trumpetG3_downoctave.wav`

  - Process: Down Octave

- `trumpetG3_halverhyth.wav`
  - Process: Halve Rhythm
- `trumpetG3_specshift.wav`
  - Process: Spectrum Quadrant Shift
- `trumpetG3_upoctave.wav`
  - Process: Up Octave

Pitch Shifting

These audio files demonstrate the results of the pitch shifting process.

- `loop120_quartwidth_up12.wav`
  - Source file: loop120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Pitch Shift: +12 semitones
- `simple120_eighthwidth_down5.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Pitch Shift: -5 semitones
- `simple120_eighthwidth_up9_psync.wav`
  - Source file: simple120.wav
  - Analysis Mode: Pitch-Synchronous Rhythmic
  - Raster Image Width: One eighth-note
  - Pitch Shift: +9 semitones
- `simple120_eighthwidth_up9.wav`
  - Source file: simple120.wav

  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Pitch Shift: +9 semitones
- `simple120_quartwidth_up9.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Pitch Shift: +9 semitones
- `trumpetG3_down7.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Pitch Shift: -7 semitones
- `trumpetG3_up5.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Pitch Shift: +5 semitones
- `trumpetG3_up36.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Pitch Shift: +36 semitones

Resize

These audio files demonstrate the results of the resizing the 2D Fourier spectrum in rhythmic analysis mode. Both signals had an original tempo of 120 bpm and a duration of 32 eighth-note beats.

- `loop120_eighthwidth_dur12.wav`
  - Source file: loop120.wav

  - Duration: 12 eighth-note beats
  - Tempo: 120 bpm
  - Quadrant Dimensions: [7 11026]
- `loop120_eighthwidth_dur24.wav`
  - Source file: loop120.wav
  - Duration: 24 eighth-note beats
  - Tempo: 120 bpm
  - Quadrant Dimensions: [13 11026]
- `loop120_eighthwidth_tempo90.wav`
  - Source file: loop120.wav
  - Duration: 16 eighth-note beats
  - Tempo: 90 bpm
  - Quadrant Dimensions: [9 14701]
- `loop120_eighthwidth_tempo150.wav`
  - Source file: loop120.wav
  - Duration: 16 eighth-note beats
  - Tempo: 150 bpm
  - Quadrant Dimensions: [9 8821]
- `simple120_eighthwidth_dur12.wav`
  - Source file: simple120.wav
  - Duration: 12 eighth-note beats
  - Tempo: 120 bpm
  - Quadrant Dimensions: [7 11026]
- `simple120_eighthwidth_dur24.wav`
  - Source file: simple120.wav

  - Duration: 24 eighth-note beats
  - Tempo: 120 bpm
  - Quadrant Dimensions: [13 11026]
- `simple120_eighthwidth_tempo90.wav`
  - Source file: simple120.wav
  - Duration: 16 eighth-note beats
  - Tempo: 90 bpm
  - Quadrant Dimensions: [9 14701]
- `simple120_eighthwidth_tempo150.wav`
  - Source file: simple120.wav
  - Duration: 16 eighth-note beats
  - Tempo: 150 bpm
  - Quadrant Dimensions: [9 8821]

Rotate

These audio files demonstrate the results of the rotating the 2D Fourier spectrum.

- `simple120_eighthwidth_90.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Rotation: 90 °
- `simple120_eighthwidth_180.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Rotation: 180 °
- `simple120_eighthwidth_270.wav`

  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Rotation: 270 °

- `trumpetG3_90.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Rotation: 90 °

- `trumpetG3_180.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Rotation: 180 °

Row Shift

These audio files demonstrate the results of processing using the row shift transformation.

- `am_bipolar_shift2remove.wav`
  This provides an excellent demonstration of rhythmic frequency. By shifting the rows of this amplitude-modulated sinusoid upwards, the modulating frequency is increased.
  - Source file: am_bipolar.wav
  - Analysis Mode: Timbral
  - Row Shift: 2
  - Remove original rows

- `loop120_eighthwidth_shift4wrap.wav`
  - Source file: loop120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note

  - Row Shift: 4
  - Wrap rows around
- `loop120_quartwidth_shift4wrap.wav`
  - Source file: loop120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Row Shift: 4
  - Wrap rows around
- `simple120_eighthwidth_shift4wrap.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Row Shift: 4
  - Wrap rows around
- `simple120_quartwidth_shift1remove.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Row Shift: 1
  - Remove original rows
- `simple120_quartwidth_shift4leave.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Row Shift: 4

  - Leave original rows

- `simple120_quartwidth_shift4wrap.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Row Shift: 4
  - Wrap rows around
- `simple120_quartwidth_shift6wrap.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One quarter-note
  - Row Shift: 6
  - Wrap rows around
- `trumpetG3_shift20_leave.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Row Shift: 20
  - Leave original rows
- `trumpetG3_shift40_remove.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Row Shift: 40
  - Remove original rows

Stretch Rhythm

These audio files demonstrate the results of processing by changing the rhythmic frequency range.

- `simple120_eighthwidth_0pt4.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Stretch factor: 0.4
- `simple120_eighthwidth_1pt5.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Stretch factor: 1.5
- `trumpetG3_0pt4.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Stretch factor: 0.4
- `trumpetG3_1pt5.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Stretch factor: 1.5

Thresh

These audio files demonstrate the results of magnitude thresholding.

- `simple120_eighthwidth_aud_10above.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Frequency Mode: Audible
  - Threshold: 10%

  - Remove: Above Threshold
- `simple120_eighthwidth_aud_80below.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Frequency Mode: Audible
  - Threshold: 80%
  - Remove: Below Threshold
- `simple120_eighthwidth_rhyth_25above.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Frequency Mode: Rhythmic
  - Threshold: 25%
  - Remove: Above Threshold
- `simple120_eighthwidth_rhyth_25below.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Frequency Mode: Audible
  - Threshold: 25%
  - Remove: Below Threshold
- `simple120_eighthwidth_single_25above.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic

  - Raster Image Width: One eighth-note
  - Frequency Mode: Single Point
  - Threshold: 25%
  - Remove: Above Threshold
- `simple120_eighthwidth_single_25below.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Frequency Mode: Single Point
  - Threshold: 25%
  - Remove: Below Threshold
- `simple120_eighthwidth_single_55below.wav`
  - Source file: simple120.wav
  - Analysis Mode: Rhythmic
  - Raster Image Width: One eighth-note
  - Frequency Mode: Single Point
  - Threshold: 55%
  - Remove: Below Threshold
- `trumpetG3_aud_25above.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Frequency Mode: Audible
  - Threshold: 25%
  - Remove: Above Threshold
- `trumpetG3_aud_63below.wav`

  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Frequency Mode: Audible
  - Threshold: 63%
  - Remove: Below Threshold
- `trumpetG3_rhyth_20above.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Frequency Mode: Rhythmic
  - Threshold: 20%
  - Remove: Above Threshold
- `trumpetG3_rhyth_20below.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Frequency Mode: Rhythmic
  - Threshold: 20%
  - Remove: Below Threshold
- `trumpetG3_single_9above.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Frequency Mode: Single
  - Threshold: 9%
  - Remove: Above Threshold
- `trumpetG3_single_20below.wav`
  - Source file: trumpetG3.wav

  - Analysis Mode: Timbral
  - Frequency Mode: Single
  - Threshold: 20%
  - Remove: Below Threshold
- `trumpetG3_single_60below.wav`
  - Source file: trumpetG3.wav
  - Analysis Mode: Timbral
  - Frequency Mode: Single
  - Threshold: 60%
  - Remove: Below Threshold